

INFORMIX®- ESQL/COBOL

Embedded SQL for COBOL

Programmer's Manual

Version 7.2
April 1996
Part No. 000-7893A

Published by INFORMIX® Press

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

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®, C-ISAM®, INFORMIX®-OnLine Dynamic Server™

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:

X/Open Company Ltd.: UNIX®, X/Open®
Adobe Systems Incorporated: PostScript®
International Business Machines Corporation: IBM®, DRDA™
Novell, Inc.: NetWare®, IPX/SPX™
Sun Microsystems, Inc.: Sun Microsystems™, NFS®
Micro Focus Ltd.: Micro Focus®, Micro Focus COBOL/2™
Ryan-McFarland (Liant) Corporation: Ryan-McFarland®

Some of the products or services mentioned in this document are provided by companies other than Informix. These products or services are identified by the trademark or servicemark of the appropriate company. If you have a question about one of those products or services, please call the company in question directly.

Documentation Team: Geeta Karmarkar, Steve Klitzing, Eileen Wollam

Copyright © 1981-1996 by Informix Software, Inc. All rights reserved.

No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

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 LEGEND

Software and accompanying materials acquired with United States Federal Government funds or intended for use within or for any United States federal agency are provided with “Restricted Rights” as defined in DFARS 252.227-7013(c)(1)(ii) or FAR 52.227-19.

Table of Contents

Introduction

About This Manual	3
Organization of This Manual	3
Types of Users	4
Software Dependencies	5
Demonstration Database	5
New Features of This Product	8
Conventions	10
Typographical Conventions	10
Icon Conventions	11
Command-Line Conventions	12
Additional Documentation	15
Printed Documentation	15
On-Line Documentation	16
Vendor-Specific Documentation	17
Related Reading	18
Compliance with Industry Standards	19
Informix Welcomes Your Comments	19

Chapter 1

Programming with INFORMIX-ESQL/COBOL

What Is INFORMIX-ESQL/COBOL?	1-3
Preparing to Use INFORMIX-ESQL/COBOL	1-4
Creating a COBOL Run-Time Program	1-6
The RM/COBOL-85 Run-Time Program	1-7
The MF COBOL/2 Run-Time Program	1-9
Using SQL in COBOL Programs	1-9
Embedding SQL Statements in COBOL Programs.	1-9
COBOL Statement Format	1-13
Including Comments.	1-18
Reserved Words and Conventions	1-18
Error Handling.	1-19

Using Host Variables in SQL Statements	1-22
Declaring Host Variables	1-22
Declaring Group Items and Arrays	1-24
Using Indicator Variables in SQL Statements	1-25
Following Rules for Indicator Variables	1-26
Representing Indicator Variables	1-27
Declaring Indicator Variables	1-27
Indicator Variables and Null Values	1-28
The INFORMIX-ESQL/COBOL Preprocessor	1-30
Supported ESQL/COBOL Preprocessor Instructions	1-30
INCLUDE Statements	1-31
Compiling INFORMIX-ESQL/COBOL Programs	1-33
The esqlcobol Command	1-34
Preprocessing, Compiling, and Linking	1-35
Preprocessor Naming Options	1-37
Running a Program	1-43
A Sample INFORMIX-ESQL/COBOL Program	1-46
The DEMO1.ECO Program	1-47
Explanation of DEMO1.ECO	1-49

Chapter 2

INFORMIX-ESQL/COBOL Data Types

Choosing Data Types for Host Variables	2-4
BINARY or COMP Data Using MF COBOL/2.	2-6
Data Conversion	2-7
Converting CHARACTER Data.	2-8
Converting SMALLINT Data	2-8
Converting INTEGER Data	2-8
Converting FLOAT, SMALLFLOAT, and DECIMAL Data	2-9
Converting DATE Data.	2-9
Data Discrepancies During Conversion	2-10
The CHAR Data Type	2-11
CHAR Type Routines	2-13
ECO-DSH	2-14
ECO-USH	2-17
ECO-GST	2-20
ECO-SQC	2-21
The VARCHAR Data Type	2-22
Data Comparison of VARCHAR Values	2-22
Programming with VARCHAR Host Variables	2-23
The TEXT and BYTE Data Types	2-25
Working with Blobs	2-25
Using Blobs with Dynamic SQL.	2-27

Numeric-Formatting Routines	2-31
Formatting Numeric Strings	2-33
ECO-FFL	2-41
ECO-FIN	2-43

Chapter 3 Working with Time Data Types

DATE Type Routines	3-3
ECO-DAT	3-7
ECO-DAY	3-10
ECO-DEF	3-13
ECO-FMT	3-18
ECO-JUL	3-23
ECO-LYR	3-26
ECO-MDY	3-28
ECO-STR	3-31
ECO-TDY	3-33
DATETIME and INTERVAL Type Routines	3-35
ANSI SQL Standards for DATETIME and INTERVAL	
Values	3-36
ECO-DAI	3-38
ECO-DSI	3-42
ECO-DTC	3-45
ECO-DTCVASC	3-47
ECO-DTS	3-52
ECO-DTTOASC	3-57
ECO-DTX	3-62
ECO-IDI	3-65
ECO-IDN	3-69
ECO-IMN	3-73
ECO-INCVASC	3-77
ECO-INTOASC	3-82
ECO-INX	3-87
ECO-IQU	3-90
ECO-SQU	3-93

Chapter 4 Error Handling

Obtaining Diagnostic Information After an SQL Statement Executes	4-4
The GET DIAGNOSTICS Statement	4-4
Statement Information	4-4
Exception Information	4-5
Examples Illustrating the GET DIAGNOSTICS Statement	4-7
Using the SQLSTATE Variable.	4-9
Multiple Error Conditions	4-16
The SQLCA Record	4-17
The Contents of the SQLCA Structure	4-19
Using SQLCODE OF SQLCA	4-21

Codes for SQL Statement Results	4-21
Success	4-22
Success with Warning	4-22
No Data Found	4-22
Error	4-24
Error Handling in Programs	4-25
Checking for Errors with the GET DIAGNOSTICS Statement	4-25
Checking for an Error Using In-Line Code	4-26
Automatically Checking for Errors Using the WHENEVER Statement	4-29
Checking for Warnings Using GET DIAGNOSTICS	4-35
Checking for Warnings Using the SQLWARN OF SQLCA Structure	4-38
ECO-MSG	4-41
A Program That Uses Full Error Checking	4-45

Chapter 5

Working with the Database Server

Understanding Database Server Connections	5-4
Client/Server Architecture of ESQL/COBOL Applications	5-4
Connecting an ESQL/COBOL Application to a Database Server	5-6
Using Callback Procedures	5-13
Routines That Work with the Database Server	5-27
ECO-SIG.	5-28
ECO-SQB	5-31
ECO-SQBCB	5-32
ECO-SQD	5-35
ECO-SQE	5-37
ECO-SQS	5-41

Chapter 6

Dynamic Management in INFORMIX-ESQL/COBOL

Programming with Dynamic SQL Statements	6-4
Working with a System Descriptor Area in INFORMIX-ESQL/COBOL	6-5
Dynamic SQL Statements and Management Techniques.	6-7
When You Need Dynamic SQL Statements	6-8
The System Descriptor Area in ESQL/COBOL	6-10
Using a System Descriptor Area	6-10

SELECT Statements That Receive WHERE-Clause Values	
at Run Time	6-19
Using Host Variables	6-20
Using a System Descriptor Area	6-21
SELECT Statements in Which Select-List Values Are	
Determined at Run Time	6-24
Non-SELECT Statements That Receive Values at Run	
Time	6-26
Using Host Variables	6-26
Using a System Descriptor Area	6-27
Non-SELECT Statements That Do Not Receive Values	
at Run Time	6-27
Using the EXECUTE IMMEDIATE Statement	6-28
Executing Stored Procedures That Receive Arguments	
at Run Time	6-29
Creating a Stored Procedure	6-30
Executing a Stored Procedure Within Your	
ESQL/COBOL Application	6-30
Dynamic SQL Program Examples	6-34
The DEMO2.ECO Program.	6-35
Explanation of DEMO2.ECO	6-39
The DEMO3.ECO Program.	6-49
Explanation of DEMO3.ECO	6-54

Appendix A List of INFORMIX-ESQL/COBOL Routines

Index

Introduction

About This Manual	3
Organization of This Manual	3
Types of Users	4
Software Dependencies	5
Demonstration Database	5
New Features of This Product	8
Conventions	10
Typographical Conventions	10
Icon Conventions	11
Comment Icons	11
Compliance Icons	12
Command-Line Conventions	12
Additional Documentation	15
Printed Documentation	15
On-Line Documentation.	16
Error Message Files	16
Release Notes, Documentation Notes, Machine Notes	17
Vendor-Specific Documentation	17
Related Reading	18
Compliance with Industry Standards	19
Informix Welcomes Your Comments	19

This chapter introduces the *INFORMIX-ESQL/COBOL Programmer's Manual*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout this manual.

About This Manual

The *INFORMIX-ESQL/COBOL Programmer's Manual* describes the INFORMIX-ESQL/COBOL SQL application-programming interface (API) that enables the COBOL programmer to create custom COBOL applications with database-management capabilities.

The *INFORMIX-ESQL/COBOL Programmer's Manual* provides information about the INFORMIX-ESQL/COBOL product and explains how to access and use the libraries, header files, and preprocessor that are provided with INFORMIX-ESQL/COBOL to enable you to access databases, manipulate the data in your program, interact with the database server, and check for errors.

Organization of This Manual

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

- This Introduction describes how INFORMIX-ESQL/COBOL fits into the Informix family of products and books, explains how to use this book, introduces the demonstration database from which the product examples are drawn, describes the Informix Messages and Corrections product, and lists the new features for Version 7.1 of INFORMIX-ESQL/COBOL.
- [Chapter 1, “Programming with INFORMIX-ESQL/COBOL,”](#) provides the basic information that you need to program in INFOR-

MIX-ESQL/COBOL. It discusses how to write, preprocess, and compile COBOL programs that contain embedded SQL statements and how to use identifiers and host variables. It includes an example that illustrates the main concepts of INFORMIX-ESQL/COBOL programming.

- [Chapter 2, “INFORMIX-ESQL/COBOL Data Types,”](#) discusses the data types that SQL recognizes and their correspondence to COBOL language data types. It also describes and illustrates data conversion and numeric-formatting routines plus character-string manipulation routines.
- [Chapter 3, “Working with Time Data Types,”](#) contains detailed descriptions of the library routines that permit the manipulation of DATE, DATETIME, and INTERVAL data types.
- [Chapter 4, “Error Handling,”](#) explains how to use error checking effectively in your INFORMIX-ESQL/COBOL programs.
- [Chapter 5, “Working with the Database Server,”](#) describes callback procedures and miscellaneous run-time routines included with the INFORMIX-ESQL/COBOL library extension, which allow you to work with the database server.
- [Chapter 6, “Dynamic Management in INFORMIX-ESQL/COBOL,”](#) discusses how INFORMIX-ESQL/COBOL handles dynamic management and illustrates two demonstration programs that contain embedded dynamic SQL statements that are provided with the software.
- [Appendix A](#) lists the INFORMIX-ESQL/COBOL library routines described throughout this manual. The list includes chapter and page references for all the routines.

Types of Users

The *INFORMIX-ESQL/COBOL Programmer's Manual* is written for all of the INFORMIX-ESQL/COBOL users and those individuals who administrate INFORMIX-ESQL/COBOL applications.

Software Dependencies

INFORMIX-ESQL/COBOL works with a database server, either INFORMIX-OnLine Dynamic Server or INFORMIX-SE, or with the INFORMIX-Enterprise Gateway product. When you compile INFORMIX-ESQL/COBOL programs, those programs can automatically communicate with database servers in a distributed network environment. You need only specify in the **sqlhosts** file the database servers with which your programs communicate. Specifying database servers in the **sqlhosts** file is described in both the [INFORMIX-SE Administrator's Guide](#) and the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#). Also, refer to [Chapter 5, "Working with the Database Server,"](#) in this manual for networking and connection information specific to INFORMIX-ESQL/COBOL.

Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are also included.

Most examples in this manual are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in Appendix A of the [Informix Guide to SQL: Reference](#).

The script that you use to install the demonstration database is called **esqlcbdemo7** and is located in the **\$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores7**. Use the following rules for naming your database:

- Names can have a maximum of 18 characters for INFORMIX-OnLine Dynamic Server databases and a maximum of 10 characters for INFORMIX-SE databases.
- The first character of a name must be a letter or an underscore (_).
- You can use letters, characters, and underscores (_) for the rest of the name.

- INFORMIX-ESQL/COBOL makes no distinction between uppercase and lowercase letters.
- The database name must be unique.

When you run **esqlcobdemo7**, you are, as the creator of the database, the owner and Database Administrator (DBA) of that database.

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

You can run the **esqlcobdemo7** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete and asks if you would like to copy the sample command files to the current directory. Enter **N** if you have made changes to the sample files and do not want them replaced with the original versions. Enter **Y** if you want to copy over the sample command files.

To create and populate the stores7 demonstration database

1. Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed.
2. Set **INFORMIXSERVER** to the name of the default database server.

The name of the default database server must exist in the **\$INFORMIXDIR/etc/sqlhosts** file. (For a full description of environment variables, see Chapter 4 of the [Informix Guide to SQL: Reference](#).) For information about **sqlhosts**, see the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#) or the [INFORMIX-SE Administrator's Guide](#).

3. Create a new directory for the SQL command files. Create the directory by entering the following command:

```
mkdir dirname
```

4. Make the new directory the current directory by entering the following command:

```
cd dirname
```

5. Create the demonstration database and copy over the sample command files by entering the **esqlcobdemo7** command.

To create the database without logging, enter the following command:

```
esqlcobdemo7 dbname
```

To create the demonstration database with logging, enter the following command:

```
esqlcobdemo7 -log dbname
```

If you are using INFORMIX-OnLine Dynamic Server, by default the data for the database is put into the root dbspace. If you wish, you can specify a dbspace for the demonstration database.

To create a demonstration database in a particular dbspace, enter the following command:

```
esqlcobdemo7 dbname -dbspace dbspacename
```

You can specify all of the options in one command, as shown in the following command:

```
esqlcobdemo7 -log dbname -dbspace dbspacename
```

If you are using INFORMIX-SE, a subdirectory called **dbname.dbs** is created in your current directory and the database files associated with **stores7** are placed there. You will see both data (**.dat**) and index (**.idx**) files in the **dbname.dbs** directory. (If you specify a dbspace name, it is ignored.)

To use the database and the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **esqlcobdemo7** script. Check with your system administrator for more information about operating-system file and directory permissions. UNIX permissions are discussed in the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#) and the [INFORMIX-SE Administrator's Guide](#).

6. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command.
7. To give someone else access to the database that you have created, grant them the appropriate privileges using the GRANT statement.

To revoke privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in Chapter 1 of the [Informix Guide to SQL: Syntax](#).

New Features of This Product

The Introduction to each Version 7.2 product manual contains a list of new features for that product. The Introduction to each manual in the Version 7.2 *Informix Guide to SQL* series contains a list of new SQL features.

A comprehensive list of all of the new features for Version 7.2 Informix products is in the Release Notes file called **SERVERS_7.2**.

This section highlights the major new features implemented in Version 7.2 of INFORMIX-ESQL/COBOL.

- Support for Level 88 variables

When you declare variables in the working storage section, you can declare those variables as Level 88. See Chapter 1 for details.

- Support for the COMP-2 specifier when using the MF COBOL/2 compiler

When you use the MF COBOL/2 compiler, you can specify COMP-2 in your variable declarations. See Chapter 2 for details.

- Support for Global Language Support (GLS)

The following INFORMIX-ESQL/COBOL routines support some GLS functionality:

- ❑ ECO-DSH
- ❑ ECO-USH
- ❑ ECO-FIN
- ❑ ECO-FFL
- ❑ ECO-DEF
- ❑ ECO-DAT
- ❑ ECO-FMT
- ❑ ECO-STR
- ❑ ECO-DTTOASC

See Chapters 2 and 3 for details.

SQLWARN7 supports locale-specific warnings. See [Chapter 4](#) for details.

Host variables and indicator variables support locale-specific language. See the [Guide to GLS Functionality](#) for details. ♦

GLS

- SQLWARN1 and SQLWARN3 generate warnings under additional conditions. See Chapter 4 for details.
- The following SQL statements are new but retain the constant values of the GRANT and REVOKE statements that they replace:
 - GRANT FRAGMENT
 - REVOKE FRAGMENT

See Chapter 6 for details.

- The following SQL statements are new and have new constant values:
 - SET
 - START VIOLATIONS TABLE
 - STOP VIOLATIONS TABLE

See Chapter 6 for details.

- Support for the year 2000.

INFORMIX-ESQL/COBOL allows you to use the year 2000 when you provide two-digit year values under the following conditions:

- String-to-date conversions
- String-to-datetime conversions
- Date literals
- Datetime literals

For more information on support for the year 2000, see Chapter 3 and the [*Informix Guide to SQL: Reference*](#).

Conventions

This section describes the conventions that are used in this manual. By becoming familiar with these conventions, you will find it easier to gather information from this and other volumes in the documentation set.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions

Typographical Conventions

This manual uses a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout this manual.

Convention	Meaning
<i>italics</i>	Within text, new terms and emphasized words are printed in italics. Within syntax diagrams, values that you are to specify are printed in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, table names, column names, menu items, command names, and other similar terms are printed in boldface.
<code>monospace</code>	Information that the product displays and information that you enter are printed in a monospace typeface.
KEYWORD	All keywords appear in uppercase letters.
◆	This symbol indicates the end of product- or platform-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 three types of information, as described in the following table. This information is always displayed in *italics*.

Icon	Description
	Identifies paragraphs that contain vital instructions, cautions, or critical information.
	Identifies paragraphs that contain significant information about the feature or operation that is being described.
	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
 ANSI	Identifies information that is specific to an ANSI-compliant database.
 GLS	Identifies information that is specific to a GLS-compliant database or application.

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 compliance information.

Command-Line Conventions

INFORMIX-ESQL/COBOL supports a variety of command-line options. You enter these commands at the operating-system prompt to perform certain functions in INFORMIX-ESQL/COBOL. Each valid command-line option is illustrated in a diagram in [Chapter 1, “Programming with INFORMIX-ESQL/COBOL.”](#)


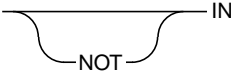
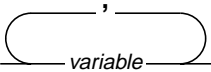
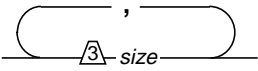
This section defines and illustrates the format of the commands that are available in INFORMIX-ESQL/COBOL and other Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

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

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 must use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. 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.
.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 and a period. 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.
<div>Privileges p. 5-17</div> <div>Privileges</div>	A reference in a box represents a subdiagram on the same page (if no page is supplied) or another page. Imagine that the subdiagram is spliced into the main diagram at this point.
— ALL —	A shaded option is the default. If you do not explicitly type the option, it will be in effect unless you choose another option.
→ →	Syntax enclosed in a pair of arrows indicates that this is a subdiagram.

(1 of 2)

Element	Description
	The vertical line is a terminator and indicates that the statement is complete.
	A branch below the main line indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items, as in this example.
	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)

Figure 1 shows how you read the command-line diagram for setting the **INFORMIXC** environment variable.

Figure 1
An Example Command-Line Diagram



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

To read the example command-line diagram

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 ENTER to execute the command.

Additional Documentation

The *INFORMIX-ESQL/COBOL Programmer's Manual* documentation set includes printed manuals, on-line manuals, and on-line help.

This section describes the following pieces of the documentation set:

- Printed documentation
- On-line documentation
- Vendor-specific documentation
- Related reading

Printed Documentation

The following printed manuals are included in the INFORMIX-ESQL/COBOL documentation set:

- If you have never used Structured Query Language (SQL), read the [*Informix Guide to SQL: Tutorial*](#). It provides a tutorial on SQL as it is implemented by Informix products. It also describes the fundamental ideas and terminology for planning, implementing, and using a relational database.
- A companion volume to the Tutorial, the [*Informix Guide to SQL: Reference*](#), includes details of the Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the column data types that Informix database servers support.
- An additional companion volume to the Reference, the [*Informix Guide to SQL: Syntax*](#), provides a detailed description of all the SQL statements supported by Informix products. This guide also provides a detailed description of Stored Procedure Language (SPL) statements.
- The [*SQL Quick Syntax Guide*](#) contains syntax diagrams for all statements and segments described in this manual.
- You, or whoever installs your Informix products, should refer to the [*UNIX Products Installation Guide*](#) for your particular release to ensure that your Informix product is properly set up before you begin to

work with it. A matrix depicting possible client/server configurations is included in the [UNIX Products Installation Guide](#).

The following related Informix documents complement the information in this manual set:

- Depending on the database server you are using, you or your system administrator need either the [INFORMIX-SE Administrator's Guide](#) or the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#).
- The [DB-Access User Manual](#) describes how to invoke the DB-Access utility to access, modify, and retrieve information from Informix database servers.
- When errors occur, you can look them up by number and learn their cause and solution in the [Informix Error Messages](#) manual. If you prefer, you can look up the error messages in the on-line message file described in “[Error Message Files](#)” later in this Introduction and in the Introduction to the [Informix Error Messages](#) manual.

On-Line Documentation

Several different types of on-line documentation are available:

- On-line error messages
- Release notes, documentation notes, and machine notes

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. 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**). See the Introduction to the [Informix Error Messages](#) manual for a detailed description of these scripts.

The optional Informix Messages and Corrections product provides PostScript files that contain the error messages and their corrective actions. If you have installed this product, you can print the PostScript files on a PostScript printer. The PostScript error messages are distributed in a number of files of the format **errmsg1.ps**, **errmsg2.ps**, and so on. These files are located in the **\$INFORMIXDIR/msg** directory.

Release Notes, Documentation Notes, Machine Notes

In addition to the Informix set of manuals, the following on-line files, located in the **\$INFORMIXDIR/release/en_us/0333** directory, might supplement the information in this manual.

On-Line File	Purpose
Documentation notes	Describes features that are not covered in the manuals or that have been modified since publication. The file that contains documentation notes for this product is called ESQLCOBDOC_7.2 .
Release notes	Describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds. The file that contains the release notes for this product is called SERVERS_7.2 .
Machine notes	Describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product that is described. The file that contains the machine notes for this product is called ESQLCOB_7.2 .

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

Vendor-Specific Documentation

For more information about the MF COBOL/2 and RM/COBOL-85 COBOL compilers used with INFORMIX-ESQL/COBOL, refer to the documentation provided with those compilers.

Related Reading

For additional technical information on database management, consult the following books. The first book is an introductory text for readers who are new to database management, while the second book is a more complex technical work for SQL programmers and database administrators:

- *Database: A Primer* by C. J. Date (Addison-Wesley Publishing, 1983)
- *An Introduction to Database Systems* by C. J. Date (Addison-Wesley Publishing, 1994).

To learn more about the SQL language, consider the following books:

- *A Guide to the SQL Standard* by C.J. Date with H. Darwen (Addison-Wesley Publishing, 1993)
- *Understanding the New SQL: A Complete Guide* by J. Melton and A. Simon (Morgan Kaufmann Publishers, 1993)
- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

The *INFORMIX-ESQL/COBOL Programmer's Manual* assumes that you are familiar with your computer operating system. If you have limited UNIX system experience, consult your operating-system manual or a good introductory text before you read this manual. The following texts provide a good introduction to UNIX systems:

- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *Learning the UNIX Operating System*, by G. Todino, J. Strang, and J. Peek (O'Reilly & Associates, 1993)
- *A Practical Guide to the UNIX System*, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)
- *UNIX System V: A Practical Guide* by M. Sobell (Benjamin/Cummings Publishing, 1995)

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-OnLine Dynamic Server. In addition, many features of OnLine comply with the SQL-92 Intermediate and Full Level and X/Open CAE (common applications environment) standards.

Informix SQL-based products are compliant with ANSI SQL-92 Entry Level (published as ANSI X3.135-1992) on INFORMIX-SE with the following exceptions:

- Effective checking of constraints
- Serializable transactions

Informix Welcomes Your Comments

Please let us know what you like or dislike about our manuals. To help us with future versions of our manuals, please tell us about any corrections or clarifications that you would find useful. 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 electronic mail, our address is:

`doc@informix.com`

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

415-926-6571

Please 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

We appreciate your feedback.

Programming with INFORMIX-ESQL/COBOL

What Is INFORMIX-ESQL/COBOL?	1-3
Preparing to Use INFORMIX-ESQL/COBOL	1-4
Creating a COBOL Run-Time Program	1-6
The RM/COBOL-85 Run-Time Program	1-7
The MF COBOL/2 Run-Time Program.	1-9
Using SQL in COBOL Programs	1-9
Embedding SQL Statements in COBOL Programs	1-9
COBOL Statement Format	1-13
Defining Area A	1-13
Defining Area B	1-15
Observing Additional Limitations	1-16
Including Comments	1-17
Reserved Words and Conventions	1-17
Error Handling	1-18
Error Handling Using GET DIAGNOSTICS and SQLSTATE	1-18
Error Handling Using the SQLCA Record	1-19
Error Handling and the WHENEVER Statement	1-20
Using Host Variables in SQL Statements	1-21
Declaring Host Variables	1-21
Declaring Group Items and Arrays	1-23
Using Indicator Variables in SQL Statements	1-24
Following Rules for Indicator Variables	1-25
Representing Indicator Variables.	1-26
Declaring Indicator Variables	1-26
Indicator Variables and Null Values.	1-27
Generating Compiler Errors After Your Program	
Returns Null Values	1-28
Inserting a Null Value Using a Negative Indicator Variable	1-28

The INFORMIX-ESQL/COBOL Preprocessor	1-29
Supported ESQL/COBOL Preprocessor Instructions	1-29
INCLUDE Statements	1-30
Compiling INFORMIX-ESQL/COBOL Programs	1-32
The esqlcobol Command	1-33
Preprocessing, Compiling, and Linking	1-34
Preprocessing Only	1-35
Displaying the Processing Steps	1-35
Preprocessor Naming Options.	1-36
Checking the Version Number	1-37
Including an Alternative SQLCA Header File	1-38
Checking for ANSI-Standard Syntax	1-39
Checking for Missing Indicator Variables	1-40
Compiling in X/Open Mode	1-41
Redirecting Errors and Warnings	1-41
Limiting the Scope of Cursor Names and Statement Ids	1-41
Defining and Undefined Values While Preprocessing	1-42
Running a Program	1-42
A Sample INFORMIX-ESQL/COBOL Program	1-45
The DEMO1.ECO Program	1-46
Explanation of DEMO1.ECO	1-48

T

his chapter introduces INFORMIX-ESQL/COBOL and tells you what you must do to begin working with ESQL/COBOL. It also describes the structure of an ESQL/COBOL program and introduces basic concepts and procedures. This chapter discusses the following topics:

- Setting up INFORMIX-ESQL/COBOL
- Preparing to use INFORMIX-ESQL/COBOL
- Creating a COBOL run-time program
- Embedding SQL statements in COBOL programs
- Using host and indicator variables in SQL statements
- Preprocessing INFORMIX-ESQL/COBOL programs
- Compiling and running INFORMIX-ESQL/COBOL programs
- An annotated sample INFORMIX-ESQL/COBOL program

What Is INFORMIX-ESQL/COBOL?

INFORMIX-ESQL/COBOL is an SQL application programming interface (SQL API) that lets you embed SQL statements directly into COBOL code. ESQL/COBOL consists of a code preprocessor, data type definitions, and COBOL routines that you can call.

An ESQL/COBOL program can use both *static* and *dynamic* SQL statements. When you use a static SQL statement, the program knows all the components at compile time. When you use a dynamic SQL statement, the program does not know all the components at compile time and instead receives all or part of the SQL statement at run time. Refer to [Chapter 6, “Dynamic Management in INFORMIX-ESQL/COBOL,”](#) for a description of dynamic SQL.



Your ESQL/COBOL program connects locally or across a network to a database server. The database server receives SQL statements from your ESQL/COBOL program, parses those statements, retrieves the requested data from a database, and sends that data back to your ESQL/COBOL program. For more information, refer to [Chapter 5, “Working with the Database Server,”](#) for a description of ESQL/COBOL client-server architecture and database server connections.

Important: *INFORMIX-ESQL/COBOL does not support multithreading.*

Preparing to Use INFORMIX-ESQL/COBOL

Before you begin working with INFORMIX-ESQL/COBOL, perform the following steps:

1. Start your computer.
Make sure your computer runs properly and an operating system prompt appears on your screen.
2. Install your Informix database server.
Make sure you install either the INFORMIX-OnLine Dynamic Server or INFORMIX-SE database server on your computer according to the installation instructions that come with your database server software. For information about the INFORMIX-OnLine Dynamic Server, refer to the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#). For information about the INFORMIX-SE database server, refer to the [INFORMIX-SE Administrator's Guide](#).
3. Install your INFORMIX-ESQL/COBOL product.
Make sure you install ESQL/COBOL on your computer according to the installation instructions that come with the software. Refer to the [UNIX Products Installation Guide](#) for those instructions.



4. Install your COBOL compiler.

Make sure you install a supported COBOL compiler (Micro Focus [MF] COBOL/2 or Ryan-McFarland [Liant] RM/COBOL-85) on your system.

Important: *Liant now owns Ryan-McFarland, thus Informix supports the Liant COBOL compiler also known as RM/COBOL-85.*

5. Set the UNIX and Informix environment variables.

Make sure you set the ESQL/COBOL environment variables shown in the following list. Refer to the [Informix Guide to SQL: Reference](#) for complete details of how to set and use the most common UNIX and Informix environment variables.

- ❑ The **INFORMIXDIR** environment variable specifies the directory where you install your database server files and your INFORMIX-ESQL/COBOL files.
- ❑ The **PATH** environment variable determines the UNIX search path for your executable programs. Make sure the pathname includes **\$INFORMIXDIR/bin** and the COBOL compiler executable.
- ❑ **INFORMIXCOB** specifies the program name of the COBOL compiler that you use. (Refer to your COBOL system manual for the name of your COBOL compiler.)
- ❑ **INFORMIXCOBDIR** contains the directory where the COBOL run-time library and/or objects reside. Use this environment variable only when you create a COBOL run-time program.
- ❑ **INFORMIXCOBSTORE** identifies the type of storage to use during compilation in an MF COBOL/2 environment. This environment variable enables ESQL/COBOL to allow or disallow certain PICTURE clauses used for mapping to internal C variable types.
- ❑ **INFORMIXCOBTYPE** contains a multicharacter code that identifies the manufacturer of the COBOL compiler that you use.
- ❑ **DBTIME** allows you to support non-ASCII datetime format specifications. When you plan to call the DATETIME manipulation routines ECO-DTCVASC or ECO-DTTOASC, make sure you set the **DBTIME** environment variable in advance.

GLS

6. Set the Global Language Support (GLS) environment variables (optional).

INFORMIX-ESQL/COBOL Version 7.2 supports GLS. Setting GLS environment variables allows you to choose whether to activate GLS capability and specify certain behaviors.

For specific information on all GLS environment variables and how to set them, refer to Chapter 2 of the [Guide to GLS Functionality](#). ♦

INFORMIX-ESQL/COBOL Version 7.2 still supports Native Language Support (NLS). For information on Native Language Support, see Informix Version 7.1 documentation.

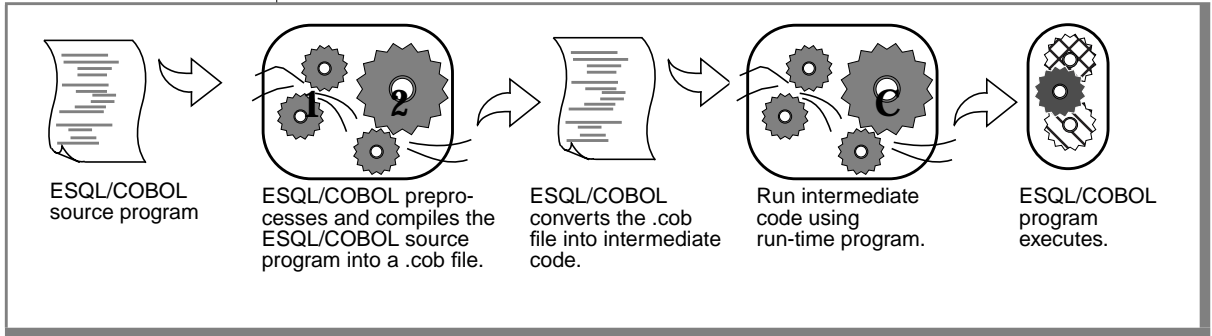
Creating a COBOL Run-Time Program

You must create a COBOL run-time program as soon as you install the INFORMIX-ESQL/COBOL product. You need to create this run-time program only once. You need a C compiler and its associated libraries to create the run-time program.

To create a COBOL run-time program for INFORMIX-ESQL/COBOL, follow the instructions for your Ryan-McFarland or Micro Focus COBOL compiler, that you find discussed on the following pages. The exact procedure can vary depending on the version of the COBOL compiler you install, your particular computer, and your particular platform.

To create an ESQL/COBOL program, write a COBOL program that includes SQL statements. You then preprocess your code using the ESQL/COBOL preprocessor. The ESQL/COBOL preprocessor takes your code, reads all of the embedded SQL statements, and generates COBOL code. The **esqlcobol** script compiles the COBOL code and converts that code into intermediate code. Figure 1-1 shows how an ESQL/COBOL program becomes an executable program.

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



Warning: The COBOL code that the INFORMIX-ESQL/COBOL preprocessor generates can change from one release of the product to the next. Therefore, make sure your ESQL/COBOL programs do not depend on the functionality and features of the product as implemented in the generated COBOL code. Rather, develop your programs according to the functionality and features of the product as described in this manual.

Refer to “[Compiling INFORMIX-ESQL/COBOL Programs](#)” on page 1-32 in this manual for information on how to preprocess and compile INFORMIX-ESQL/COBOL code to execute with this run-time program.

The RM/COBOL-85 Run-Time Program

The following procedure creates a customized COBOL run-time program for use with INFORMIX-ESQL/COBOL and RM/COBOL-85.

1. Choose a destination directory for the new run-time program and make that directory the current directory. The destination directory represents the directory where you want your application to reside.
2. Copy the **sub.c** file to the current directory. (You might find **sub.c** in the same directory where COBOL was installed on your system. Contact your system administrator for the exact location.)

3. Add the four statements, indicated with an arrow in the following example, to the **sub.c** file in the current directory:

```
.
.
.
/*
 * Define all routines as integer
 */
static int  subsys();
static int  subren();
static int  subdel();
static int  subtimes();
static int  random();
static int  srandom();
#include "rdsdec.c" <=====

struct PROCTABLE LIBTABLE[] =
{
    {"SYSTEM",subsys},
    {"RENAME",subren},
    {"DELETE",subdel},
    {"TIMES",subtimes},
    {"RAND",random},
    {"SRAND",srandom},
#include "rdsproc.c" <=====
    {0,0}
};

.
.
.

#include "cobref.h" <==== at end of file
#include "rsubrm85.c" <==== at end of file
```

4. Compile **sub.c** with the following command:

```
makerun
```

The preceding command creates a new COBOL run-time program, called **runcobol**, in the current directory.

The MF COBOL/2 Run-Time Program

To use MF COBOL/2 with INFORMIX-ESQL/COBOL, you do not need to customize the run-time program. Enter the following command to create the COBOL run-time program **newrun** in the current directory:

```
makerun newrun
```

The program name **newrun** can represent any valid name that you specify. When you do not specify a run-time program file name, ESQL/COBOL creates a default run-time program called **newrun**.

Using SQL in COBOL Programs

As a COBOL programmer, you must acquaint yourself with the following topics, discussed in this section, to most effectively use SQL statements in your COBOL programs:

- Embedding SQL statements in COBOL programs
- COBOL statement format
- Including comments
- Reserved words and conventions
- Error handling

Make sure you also read the subsequent sections of this chapter that discuss how to use host variables and indicator variables.

Embedding SQL Statements in COBOL Programs

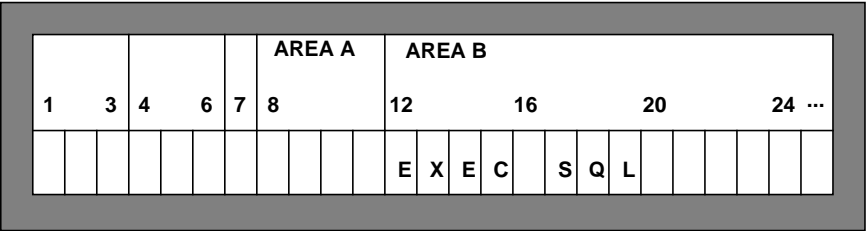
You embed SQL statements in COBOL programs in the PROCEDURE DIVISION with the EXEC SQL and END-EXEC keywords. The following full COBOL sentence includes the phrases EXEC SQL and END-EXEC:

```
EXEC SQL  
    SQL-statement  
END-EXEC.
```

Certain rules apply when you embed SQL statements in COBOL programs:

- All SQL statements, including the EXEC SQL and END-EXEC directives, must reside in Area B of the COBOL program line, as shown in Figure 1-2.

Figure 1-2
ESQL/COBOL Program-Line Fragment



- Whereas SQL statements can reside over multiple lines within Area B, you cannot split the phrases EXEC SQL or END-EXEC over multiple lines.

[Figure 1-3](#) shows correct and incorrect syntax for EXEC SQL and END-EXEC.

Figure 1-3
ESQL/COBOL Examples Showing Correct and Incorrect SQL Statement Syntax

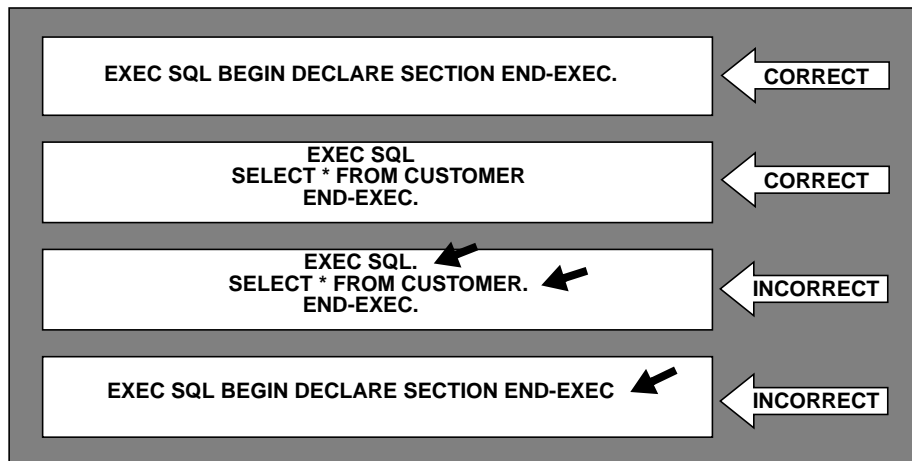
EXEC SQL	CORRECT
EXEC SQL BEGIN DECLARE SECTION END-EXEC.	CORRECT
EXEC SQL BEGIN DECLARE SECTION END-EXEC.	CORRECT
END-EXEC.	CORRECT
EXEC SQL	INCORRECT
EXEC SQL BEGIN DECLARE SECTION END- EXEC.	INCORRECT
END- EXEC.	INCORRECT

- You must terminate the statements with appropriate COBOL sentence punctuation in that location. (You cannot use a space as a terminator.) Include a period only when the EXEC SQL block terminates a COBOL sentence.

Figure 1-4 shows correct and incorrect terminator syntax for SQL statements.

Figure 1-4

ESQL/COBOL Examples Showing Correct and Incorrect SQL Terminator Syntax



In your INFORMIX-ESQL/COBOL program, you can use host variables and indicator variables anywhere that you can use a constant. For more information on host variables, refer to [“Using Host Variables in SQL Statements” on page 1-21](#) in this manual. For more information on indicator variables, refer to [“Using Indicator Variables in SQL Statements” on page 1-24](#) in this manual. Refer to individual SQL statements in the [Informix Guide to SQL: Syntax](#) for any exceptions.

COBOL Statement Format

A COBOL source program must exist in standard COBOL reference format so INFORMIX-ESQL/COBOL can process it. In other words, columns 1 through 6 of each input line must either contain blanks or a line number, and column 7 must contain the COBOL comment and continuation indicator column. The remainder of the COBOL statement resides in *Area A* and *Area B*.

Defining Area A

Area A spans columns 8 through 11 of the statement, as shown in Figure 1-5.

Figure 1-5
ESQL/COBOL Program-Line Fragment Showing Area A

						AREA A					AREA B												
1	3	4	6	7	8		11	12		16		20		24	...								
						I	D	E	N	T	I	F	I	C	A	T	I	O	N		D	I	V

- In the WORKING-STORAGE SECTION of the DATA DIVISION, only 01-level items (record definitions) and 77-level items can begin in Area A, as shown in Figure 1-6.

						AREA A			AREA B															
1	3	4	6	7	8	11	12	16	20	24	...													
							W	O	R	K	I	N	G	-	S	T	O	R	A	G	E		S	E
							7	7			F	N	A	M	E					P	I	C		X
							0	1			E	R	R	-	R	E	C			P	I	C		X

- In the PROCEDURE DIVISION, only procedure names can begin in Area A, as shown in Figure 1-7.

Figure 1-7
*ESQL/COBOL Program-Line Fragment Showing Area A
in Relation to the Procedure Division*

					AREA A		AREA B																		
1	3	4	6	7	8	11	12	16	20	24	...														
							P	R	O	C	E	D	U	R	E		D	I	V	I	S	I	O	N	
							M	A	I	N	-	P	R	O	C	E	D	U	R	E	.				
												P	E	R	F	O	R	M		O	P	E	N	-	D

Defining Area B

Area B extends from column 12 through column 72, as shown in [Figure 1-8](#). The COBOL compiler ignores columns 73 through 80. With RM/COBOL-85, when an SQL statement does not start in Area B, a compile-time warning appears.

Figure 1-8
ESQL/COBOL Program-Line Fragment Showing Area B

			AREA A	AREA B	
Cols 1-3	Cols 4-6	Col 7	Cols 8-11	Cols 12-72	*Cols 73-80
*Your COBOL compiler ignores columns 73 to 80. Specify the -bigB preprocessor option to use these columns.					

If you want to extend Area B beyond column 72, specify the **-bigB** ESQL/COBOL preprocessor option.

ESQL/COBOL reports preprocessing errors when you do not use Area A and Area B correctly.

Observing Additional Limitations

Do not use tabs within your programs. The compiler detects tabs in the first seven characters. Tabs beyond that point can create unpredictable compilation results when you write lengthy COBOL source lines or SQL statements.

The maximum line length can be 256 characters, depending on your compiler. The maximum size of a statement name is 132 characters.



Important: In INFORMIX-ESQL/COBOL, the *FILLER* keyword must identify all filler items. In RM/COBOL-85, filler items in record structures do not require the keyword *FILLER* to appear as the data name in the record declaration.

Including Comments

In addition to the standard COBOL comment indicator in column 7, you can use a double dash (--) comment indicator on any ESQL/COBOL line (one located between the EXEC SQL and END-EXEC phrases). The comment continues to the end of the line. For example, the following line specifies the **stores7** database and a comment:

```
EXEC SQL DATABASE STORES7 END-EXEC. -- STORES7 DATABASE OPENED
```

Reserved Words and Conventions

Refer to a list of reserved words specific to your operating system and the COBOL programming language when you write ESQL/COBOL code. In addition, refer to the “Identifier” segment in the [Informix Guide to SQL: Syntax](#) for a list of ANSI reserved words and SQL identifiers that can create problems when used as the name of the following database objects:

- Database
- Table
- Index
- Synonym
- Constraint

The list of reserved words also includes all words beginning with ECO- and some words beginning with SQL.

SQL statements must begin with the words EXEC SQL and end with END-EXEC. Refer to [“Embedding SQL Statements in COBOL Programs” on page 1-9](#) in this manual for details.

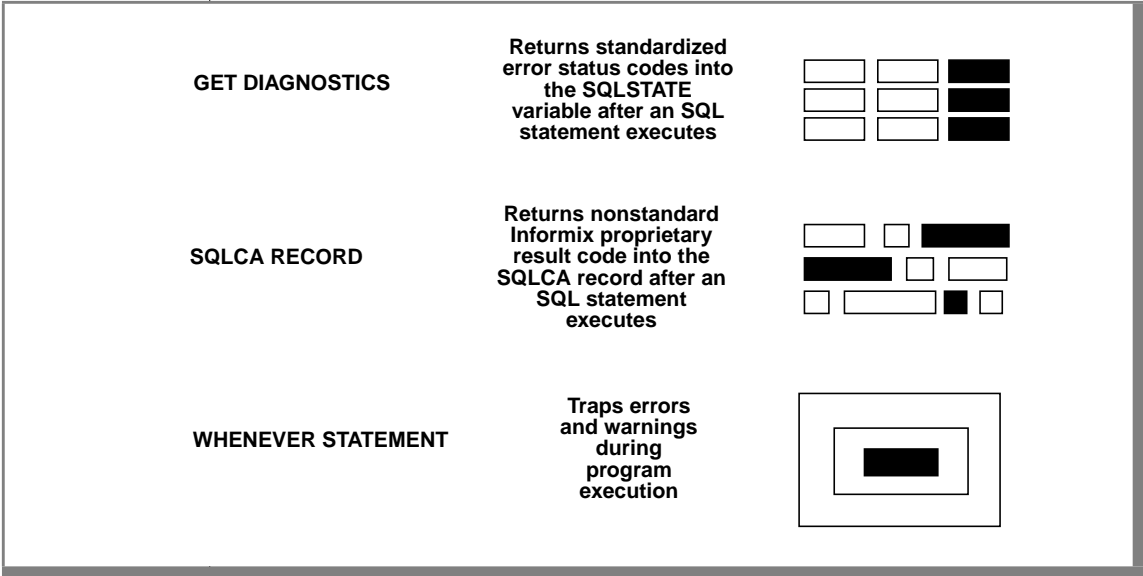
You cannot use a backslash (\) to make a single quotation mark (') transparent within a string. You must use two consecutive single quotation marks (").

The ESQL/COBOL preprocessor uses the COBOL conventions to call COBOL language subroutines from a COBOL program. (For MF COBOL/2, the preprocessor uses the numbers 0 through 99 as subroutine names. Make sure your programs do not use these numbers as paragraph or subroutine names with MF COBOL/2.)

Error Handling

Make sure every SQL statement executes correctly. You can use the GET DIAGNOSTICS statement with the SQLSTATE variable to determine the status of an SQL statement. You can also use the SQL Communications Area (SQLCA) record to determine the status of an SQL statement. Those error handling methods are compared in Figure 1-9.

Figure 1-9
ESQL/COBOL Error-Handling Methods



Error Handling Using GET DIAGNOSTICS and SQLSTATE

When you write an INFORMIX-ESQL/COBOL program to the X/Open standard, you can trap and diagnose X/Open errors using the SQLSTATE variable and the GET DIAGNOSTICS statement. When an SQL statement executes, your INFORMIX-ESQL/COBOL program generates a status code stored in the SQLSTATE variable. The value in SQLSTATE tells you that the SQL statement succeeded or failed. When an SQL statement fails, you can use the GET DIAGNOSTICS statement to gather more information on that failure. Informix recommends that you check for errors using SQLSTATE and that you diagnose errors using the GET DIAGNOSTICS statement.

The SQLSTATE error status code contains a class code and a subclass code. The following list contains the valid codes for SQLSTATE:

00000	Successful execution of the SQL statement
01000	Successful execution with a warning that the effect was limited or undesirable
02000	Successful execution but that your program found no data
Other values	The SQL statement failed

The DEMO1.ECO, DEMO2.ECO, and DEMO3.ECO programs illustrated in this manual contain an ERROR-PROCESS procedure that uses the SQLSTATE variable and the GET DIAGNOSTICS statement to handle error checking. You can find information about the syntax of GET DIAGNOSTICS in the [Informix Guide to SQL: Syntax](#). You can find complete information about the role, structure, declaration, and use of GET DIAGNOSTICS and SQLSTATE in [Chapter 4, “Error Handling,”](#) in this manual.

Error Handling Using the SQLCA Record

INFORMIX-ESQL/COBOL includes the SQLCA record in each ESQL/COBOL program automatically. The record varies, depending on the COBOL compiler you use.

INFORMIX-ESQL/COBOL returns a result code into the SQLCA record after executing every SQL statement.



Tip: Although you can use the SQLCA record to handle errors, Informix recommends that you use the GET DIAGNOSTICS statement and the SQLSTATE variable as the standard error-handling components.

The SQLCODE OF SQLCA component of the SQLCA record can contain the values shown in Figure 1-10 to indicate that an SQL statement succeeded.

Figure 1-10
Values Contained in SQLCODE of SQLCA

SQLCODE value	Description
zero	For a successful execution of SQL statements
a negative value	For an unsuccessful execution of an SQL statement
SQLNOTFOUND (100)	For a successful query that returns no rows

By taking corrective action when SQLCODE OF SQLCA contains a negative value, you can recover from the failure of an intended database modification.

By checking for SQLCODE OF SQLCA = SQLNOTFOUND, you can write your code to process the results of queries only when your program returns rows.

For more information about error handling using the SQLCA record, refer to [Chapter 4, “Error Handling.”](#)

Error Handling and the WHENEVER Statement

You can also use the WHENEVER statement in your ESQL/COBOL code to trap errors and warnings during the execution of SQL statements. When you encounter various error and warning conditions, you can include a WHENEVER statement that specifies whether the program continues, stops, or transfers control to another procedure or routine.

Refer to the [Informix Guide to SQL: Syntax](#) for a detailed understanding of the syntax and usage for the WHENEVER statement. Also refer the discussion of error handling in the [Informix Guide to SQL: Tutorial](#) and [Chapter 4, “Error Handling.”](#)

Using Host Variables in SQL Statements

Host variables represent normal COBOL variables that you use in SQL statements. They are not case-sensitive in COBOL. When you use a host variable in an SQL statement, precede its name with a colon (:) or a dollar sign (\$). The host variable HOSTVAR, for example, can appear in an SQL statement as either :HOSTVAR or \$HOSTVAR.

If you must qualify the host variable name because it appears more than once in WORKING-STORAGE, use normal COBOL qualification syntax. For example, when HOSTVAR represents part of the record name HOSTREC, make sure you represent HOSTVAR in an SQL statement as :HOSTVAR OF HOSTREC or \$HOSTVAR OF HOSTREC.

INFORMIX-ESQL/COBOL supports level 01 through level 49 COBOL variables and record types. The product also supports COBOL variable levels 77 and 88, provided that your COBOL compiler supports those variable levels.

You associate host variables with SQL data types because those variables appear in SQL statements. For a discussion of host variable data types and their correspondence with SQL column data types, refer to [“Choosing Data Types for Host Variables” on page 2-4](#) in this manual and also the [Informix Guide to SQL: Reference](#).

Declaring Host Variables

The host variable declarations reside between the statements EXEC SQL BEGIN DECLARE SECTION END-EXEC and EXEC SQL END DECLARE SECTION END-EXEC in the COBOL WORKING-STORAGE SECTION, as shown in the following example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      77 HOSTINT          PIC S9(10) COMP-5.
      77 HOSTMONEY        PIC S9(5)V99 COMP-5.
      77 HOSTFLOAT        PIC S9(6)V999 COMP-5.
      77 HOSTCHAR         PIC X(80).
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
***** OTHER NON-HOST COBOL VARIABLES *****
01  MISC-HOST-VARS.
.
.
.
```

The statements EXEC SQL BEGIN DECLARE SECTION END-EXEC and EXEC SQL END DECLARE SECTION END-EXEC must each reside on a single line.



Important: Multiple host variable declaration sections can exist in an ESQL/COBOL program.

INFORMIX-ESQL/COBOL lets you declare host variables and assign them initial values with normal COBOL VALUE clauses, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
05  VARNAME          PIC S9(04) COMP-5 VALUE -12.
05  SRCH-CUSTOMER    PIC X(20) VALUE SPACES.
05  OPENDATE         DATE_TYPE VALUE "01/15/1994".
EXEC SQL END DECLARE SECTION END-EXEC.
01  MORE-HOST-VARS.
```

GLS

The INFORMIX-ESQL/COBOL preprocessor does not check VALUE clauses for valid COBOL syntax. Instead, INFORMIX-ESQL/COBOL passes VALUE clauses to the COBOL compiler, and that compiler diagnoses any errors.

You can define as many host variables as you need, up to the limit set for the symbol table of your COBOL compiler. Only host variables (variables actually referenced in SQL statements) need to appear in the SQL DECLARE SECTION. You must declare regular program variables in the WORKING-STORAGE SECTION outside the SQL DECLARE SECTION.

INFORMIX-ESQL/COBOL does not recognize the standard COBOL features listed in [Figure 1-11](#). When you include any of the following features in a host variable declaration, the preprocessor generates a syntax error.

Figure 1-11
COBOL Clauses That ESQL/COBOL Does Not Recognize

Feature	Meaning
IS EXTERNAL	Variable available to every program in the unit that describes it
IS GLOBAL	Variable available to every program within the program that declares it
RENAMES	Creates an alternative record group
SYNCHRONIZED RIGHT	Defines memory alignment

The preceding clauses apply only to host variables. Variables declared outside EXEC SQL BEGIN DECLARE SECTION END-EXEC and EXEC SQL END DECLARE SECTION END-EXEC statements can include the clauses listed in Figure 1-11.

The language used for declaring host variables is locale-specific. The default locale for declaring host variables is U.S. ASCII English. For more locale information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Declaring Group Items and Arrays

You can declare group items as INFORMIX-ESQL/COBOL host variables in the WORKING-STORAGE SECTION. In ESQL/COBOL statements, you can use the group item name in place of the names of the component data items, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  CUST-REC.
    05  C-NO          PIC S9(08) COMP-5.
    05  FNAME         PIC X(20).
    05  LNAME         PIC X(20).
EXEC SQL END DECLARE SECTION END-EXEC.
```

The preceding declaration makes the following two code segments equivalent:

```
EXEC SQL
    INSERT INTO CUSTOMER VALUES (:CUST-REC)
END-EXEC.

EXEC SQL
    INSERT INTO CUSTOMER
    VALUES (:C-NO OF CUST-REC, :FNAME OF CUST-REC,
            :LNAME OF CUST-REC)
END-EXEC.
```

ESQL/COBOL understands and supports the declaration of arrays of variables. You can use elements of an array within ESQL/COBOL statements. For example, when you declare the following array:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
05    MONTH-NAMES      PIC X(10) OCCURS 12 TIMES.
EXEC SQL END DECLARE SECTION END-EXEC.
```

you can use the following code:

```
MOVE 1 TO MONTH-INDEX.
PERFORM LOAD-MONTHS 12 TIMES.
.
.
.
LOAD-MONTHS.
EXEC SQL
    FETCH MONTHNM_CURSOR INTO :MONTH-NAMES (MONTH-INDEX)
END-EXEC.
ADD 1 TO MONTH-INDEX.
```

GLS

Using Indicator Variables in SQL Statements

You can define *indicator variables* with host variables for the following situations:

- When your program retrieves a non-null SQL value into a host variable of CHARACTER data type, your program can truncate the value to fit into the variable.
- When a host variable of numeric data type causes a conversion error.

Following Rules for Indicator Variables

You can define indicator variables as any valid host variable data type except DATETIME or INTERVAL. You call the associated host variables *main variables*. Use the following rules for defining indicator variables:

- For variables of CHARACTER data type, make sure you define an indicator variable that ESQL/COBOL sets to the defined size of the SQL column, in bytes, before truncation (if truncation occurs).
- For variables of numeric data type, make sure you define an indicator variable that ESQL/COBOL sets to zero when no conversion error occurs, or to nonzero when a conversion error does occur.
- For character fields, when you fetch a value longer than the size of the host variable, your program sets the SQLWARN1 OF SQLWARN OF SQLCA field to W, and the indicator contains the actual length of the value in the database.
- For noncharacter fields using values fetched into character type host variables, when the character conversion yields more characters than the size of the host variable, your program sets the SQLWARN1 OF SQLWARN OF SQLCA field to W. In addition, your program fills the host variable with asterisks (*) and sets the indicator to 1.
- Your program sets an indicator variable to -1 when a query returns a null value.



Tip: For simplicity and ease of programming, Informix recommends that you use only indicator variables of the INTEGER type.

The language used for indicator variables is locale-specific. For more locale information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Representing Indicator Variables

You can represent an indicator variable in an SQL statement in either of two ways:

- Place a colon (:) between the main variable name and the indicator variable name, with no spaces. For example, when HOSTVARIND represents the indicator variable for the main variable HOSTVAR, you can represent the pair in an SQL statement as either :HOSTVAR:HOSTVARIND or \$HOSTVAR:HOSTVARIND.
- Place the INDICATOR keyword, with spaces, between the main variable name and the indicator variable name. (This use of the INDICATOR keyword conforms to the ANSI standard.) For example, when HOSTVARIND represents the indicator variable for the main variable HOSTVAR, you can represent the pair in an SQL statement as either :HOSTVAR INDICATOR :HOSTVARIND or \$HOSTVAR INDICATOR :HOSTVARIND.

Declaring Indicator Variables

The following example shows how to declare indicator variables:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    05  NAME                                PIC X(15).
    05  COMPANY-NAME                        PIC X(19).
    05  NAME-INDICATOR                      PIC S9(5) COMP-5.
    05  COMP-INDICATOR                      PIC S9(5) COMP-5.
EXEC SQL END DECLARE SECTION END-EXEC.
01  MISC-HOST-VARS.
...
PROCEDURE DIVISION.
...
    EXEC SQL
        SELECT LNAME, COMPANY
            INTO :NAME:NAME-INDICATOR,
                :COMPANY-NAME:COMP-INDICATOR
        FROM CUSTOMER
        WHERE CUSTOMER_NUM = 105
    END-EXEC.
```

If you define **lname** in the **customer** table with a length longer than 15 characters, NAME-INDICATOR contains the actual length of the **lname** column. The NAME string contains the first 15 characters. When the value of **lname** where the **customer_num** = 105 is shorter than 15 characters, the remaining characters become trailing blanks. Therefore, ESQL/COBOL truncates only trailing blanks and the host variable NAME contains all the significant data. Nevertheless, your program sets the SQLSTATE class code to 01 and sets SQLWARN0 OF SQLWARN OF SQLCA to W. When you use SQLSTATE in your program, make sure you use the GET DIAGNOSTICS statement to check for a specific error warning message.

If **company** has a null value for this same **customer_num**, then COMP-INDICATOR has a value of -1. You cannot predict the contents of the COMPANY-NAME data item.

If you set the **DBNLS** and **LANG** environment variables, you can use indicator variables in local character sets, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      05  HÔTE          PIC X(24).
...
      05  HÔTE_IND      PIC S9(10).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
```

Indicator Variables and Null Values

An SQLSTATE value of 01004 signals the occurrence of a truncation error.

The SQLCA record can also signal the occurrence of a character truncation (or a conversion error). The ESQL/COBOL library call sets SQLCODE OF SQLCA to some negative value indicating the error. When your program does not return a neither null or truncated value, the program sets the indicator variable value to zero (0).

However, the SQLSTATE value, or the value of the SQLCODE OF SQLCA, does not provide enough information to determine the host variable that generated the error. You must examine the indicator variables associated with the host variables to determine the specific host variable affected and the extent of the error or truncation.

Because a null value does not always represent a definite value among other values, you must find out when an ESQL/COBOL statement returns a null variable to a host variable. When a host variable corresponds to a database column that allows null values, you must define an indicator variable in association with that host variable.

When an ESQL/COBOL statement returns a null value to a host variable (through the INTO clause of a SELECT or FETCH statement) and you define an indicator variable, the indicator variable has a value of -1. The actual value in the host variable does not always represent a meaningful COBOL value.

Generating Compiler Errors After Your Program Returns Null Values

If you do not assign an indicator variable to the host variable and your program returns a null value, ESQL/COBOL can generate an error (or no error) depending on how you compile the program.

- If you compile the program using the **-icheck** flag, ESQL/COBOL generates an error and sets the SQLSTATE class field to a value greater than 02. It also sets SQLCODE OF SQLCA to a negative value when your program returns a null value and no indicator variable exists. (For more information, refer to [Chapter 4, “Error Handling.”](#) in this manual.)
- If you compile the program without using the **-icheck** flag, ESQL/COBOL does not generate an error when your program returns a null value and no indicator exists.

Inserting a Null Value Using a Negative Indicator Variable

As an alternative to using the NULL keyword in an INSERT statement, you can use a host variable with a negative indicator variable to place a null value in a particular column of an added row. The following example shows how to insert a null value using a negative indicator variable:

```
MOVE -1 TO ORD-SHIP-INDICATOR.  
EXEC SQL  
    INSERT INTO ORDERS (ORDER_NO, SHIP_DATE)  
    VALUES (:ORDER-NO, :ORD-SHIP-DATE:ORD-SHIP-INDICATOR)  
END-EXEC.
```

The ORD-SHIP-DATE value is null because it is not valid for a new order.

The INFORMIX-ESQL/COBOL Preprocessor

You must preprocess your programs that include ESQL/COBOL statements before you can use your COBOL compiler to compile them. The ESQL/COBOL preprocessor converts the embedded SQL statements into COBOL language code before the COBOL compiler receives them.

The ESQL/COBOL preprocessor works in the following two stages:

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

Stages 1 and 2 of the ESQL/COBOL preprocessor mirror the COBOL language preprocessor and compiler stages of compilation, as shown on [page 1-7](#). ESQL/COBOL Stage 1 preprocessor statements and COBOL preprocessor statements differ only in that the former take effect during input to the ESQL/COBOL preprocessor.

You can use Stage 1 of the ESQL/COBOL preprocessor to incorporate other files into the source file. You must include files necessary for compiling embedded SQL statements before Stage 2 of the preprocessor starts. You cannot use the normal COBOL preprocessor to conditionally compile ESQL/COBOL statements because ESQL/COBOL already processed those statements in Stage 1.

Supported ESQL/COBOL Preprocessor Instructions

The ESQL/COBOL preprocessor supports the following instructions. Use them to include files in your programs, to define values, and for conditional compilation:

- | | |
|---------|---|
| INCLUDE | includes a source file in the input at that point. |
| DEFINE | specifies a compile-time name flag. |
| UNDEF | removes an existing compile-time name flag. |
| IFDEF | tests for a name flag and executes subsequent statements only when the name flag has been defined. |
| IFNDEF | tests for a name flag and executes subsequent statements only when the name flag has <i>not</i> been defined. |

ELSE	begins an alternative section to an IFDEF or IFNDEF condition.
ELIF	begins an alternative section to an IFDEF or IFNDEF condition that checks for the presence of another IFDEF.
ENDIF	closes an IFDEF or IFNDEF condition.

Each statement used as an instruction for the ESQL/COBOL preprocessor must begin with the keywords EXEC SQL and end with the keywords END-EXEC, as shown in the following example:

```
EXEC SQL DEFINE MAXROWS 25 END-EXEC.
```

The DEFINE statement can define only INTEGER constants. It does not support definition of string constants or parameterized macros.

The preprocessor does not support a generalized IF statement, only the IFDEF and IFNDEF statements that test whether a name has been defined. In the following example, ESQL/COBOL compiles the BEGIN WORK statement only after you define the name USE-TRANSACTIONS:

```
EXEC SQL DEFINE MAXROWS 25 END-EXEC.  
EXEC SQL IFDEF USE-TRANSACTIONS END-EXEC.  
EXEC SQL BEGIN WORK END-EXEC.  
EXEC SQL ENDIF END-EXEC.
```

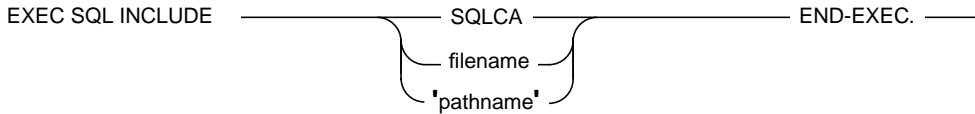
The following example shows how to use ELIF:

```
EXEC SQL IFDEF ONLINE END-EXEC.  
EXEC SQL CREATE DATABASE NEWSTORE WITH LOG END-EXEC.  
EXEC SQL ELIF SE END-EXEC.  
EXEC SQL CREATE DATABASE TMPSTORE WITH LOG IN '/tmp/log' END-  
EXEC.  
EXEC SQL ENDIF END-EXEC.
```

INCLUDE Statements

You can use the EXEC SQL INCLUDE preprocessor statement to include other ESQL/COBOL files in your program. You declare an INCLUDE statement in the data division. You can nest EXEC SQL INCLUDE statements to a depth of eight.

The EXEC SQL INCLUDE statement allows you to include files that contain only SQL statements. The following diagram illustrates the syntax of the EXEC SQL INCLUDE statement.



filename represents the name of the file that you want to include.
pathname represents the full and complete pathname of a file you want to include.
 SQLCA includes the SQLCA file.

You can write an INCLUDE statement using the name of the file or the pathname. You must use quotation marks around the pathname. The following syntax examples show how you specify a file name or a pathname in the EXEC SQL INCLUDE preprocessor statement:

```
EXEC SQL INCLUDE filename END-EXEC.
```

```
EXEC SQL INCLUDE 'pathname' END-EXEC.
```

When you use the first form (with no quotes), the preprocessor looks for the included file in the following sequence:

1. In the current directory
2. In the **\$INFORMIXDIR/incl/esql** directory, where **\$INFORMIXDIR** represents the Informix installation directory (Refer to the discussion of environment variables in the [Informix Guide to SQL: Reference](#) for more information on **INFORMIXDIR**.)
3. In the **/usr/include** directory



Important: The **-Ipathname** preprocessor option, described in the following section, expands the search range for **INCLUDE** directories. The preprocessor also searches the current directory, **pathname**, **\$INFORMIXDIR/incl/esql**, and **/usr/include**.

The INFORMIX-ESQL/COBOL preprocessor puts the included file in the current file at the position of the INCLUDE statement. Then the preprocessor processes that file and passes it on to the COBOL compiler.

You can use the INCLUDE statement in a way similar to the COBOL statement COPY. INCLUDE obtains common code from the COBOL source-statement library at compile time. However, INCLUDE brings in statements from the requested file *before* INFORMIX-ESQL/COBOL processes your current file.

The source files can contain COBOL COPY statements in addition to ESQL/COBOL INCLUDE statements. Data descriptions imported using COBOL COPY statements do not affect ESQL/COBOL statements but take effect in their usual way when the COBOL compiler receives the source file.

Compiling INFORMIX-ESQL/COBOL Programs

This section discusses how to use the ESQL/COBOL preprocessor, how to execute the object files that the compiler creates, and how to link ESQL/COBOL libraries to your own libraries.

After you preprocess your programs that include ESQL/COBOL statements, you can compile the resulting source file with your COBOL compiler to create an object file. Then, to execute the object file, invoke a COBOL run-time program that you modified to include the INFORMIX-ESQL/COBOL libraries. (Refer to “[Creating a COBOL Run-Time Program](#)” on page 1-6.)

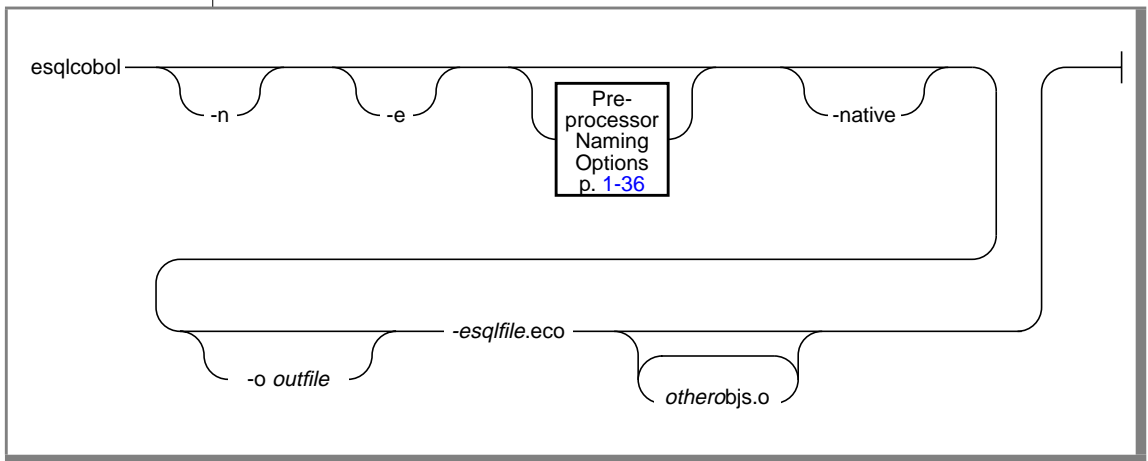
To preprocess and compile a COBOL program that contains ESQL/COBOL statements, enter `esqlcobol` on the command line, provide the name of the source file (a name that must end with the `.eco` extension), and include any arguments. The **esqlcobol** shell script invokes the ESQL/COBOL preprocessor.

INFORMIX-ESQL/COBOL establishes communications and network connections during compilation. For more information on how ESQL/COBOL establishes communication and network connections, refer to the [INFORMIX-SE Administrator's Guide](#) or the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#).

INFORMIX-ESQL/COBOL supports NLS. However, some COBOL compilers reject non-English identifiers in NLS mode because of totally incompatible underlying code sets. This problem can occur when you attempt to compile code previously processed using a different LANG setting.

The esqrcobol Command

The following pages illustrate and describe the syntax and options for preprocessor and compiler commands.



- e** performs only the preprocessor step. It produces a *pure* COBOL source file with the extension **.cob** or **.cbl** (RM/COBOL-85).
- esqfile.eco** the name of the source file that contains your ESQL/COBOL program and COBOL code. When you omit the **.eco** extension, ESQL/COBOL does not compile the program.
- n** displays on the screen the steps of the preprocessor/compilation process but does not perform them.
- native** generates native code, provided that your compiler can create a native executable file instead of an intermediate file.

-o outfile	specifies the next argument as program name (compiler-specific)
otherobjs.o	represents C objects that you want to include in your program. You must first use the -native option to include C objects when you compile your program. For more information on including C objects, your MF COBOL/2 compiler documentation.

When you enter `esqlcobol` on the command line without an argument, the supported options display on the screen. However, the option **-esqlargs** represents the preprocessor naming options listed on [page 1-36](#). You do not type **-esqlargs** to use these options. Instead, you type the option or options you want to invoke.

Preprocessing, Compiling, and Linking

In the `esqlcobol` command-line syntax, *outfile* represents the name of the intermediate or executable output file and *esqlfile.eco* represents your program source file that includes both COBOL and SQL statements. When you set the **INFORMIXDIR** environment variable correctly and no other processing anomalies occur, processing occurs in the following sequence:

1. The `esqlcobol` command executes any precompile instructions and preprocesses the embedded SQL statements in *esqlfile.eco*.
2. The `esqlcobol` command then produces an ASCII file of COBOL statements named *esqlfile.cob* or *esqlfile.cbl* (RM/COBOL-85).
3. The `esqlcobol` command then passes *esqlfile.cob*, *esqlfile.cbl*, or another COBOL source file straight through to the COBOL compiler. It produces interpreted files that require the extension **.exe** (MF COBOL/2) or the extension **.INT** (RM/COBOL-85).
4. ESQL/COBOL then links these files with the appropriate ESQL/COBOL library routines along with other COBOL files and any other system libraries that you explicitly include on the `esqlcobol` command line.

You can include a variety of preprocessor naming options and/or compiling and linking options in your command line.

Preprocessing Only

You can preprocess your ESQL/COBOL program without compiling and linking. To preprocess your code, include the **-e** flag in the **esqlcobol** command. The preprocessor creates a COBOL program. For example, to preprocess the program that resides in the file **demo1.eco**, use the following command:

```
esqlcobol -e demo1.eco
```

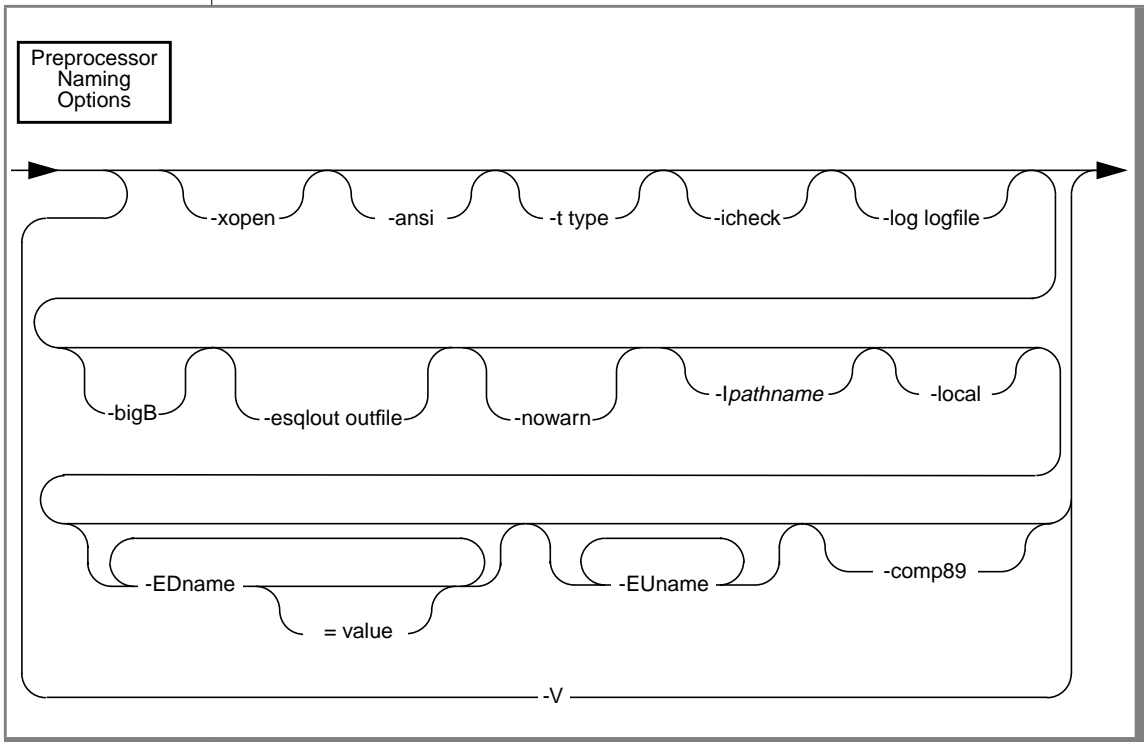
Displaying the Processing Steps

Use the **-n** flag to display the steps for preprocessing and compiling without actually executing them, as shown in the following example:

```
esqlcobol -n demo1.eco
```

Preprocessor Naming Options

The following diagram shows the syntax for the **esqlcobol** preprocessor naming options.



- ansi** checks for Informix extensions to ANSI-standard SQL syntax and sends warning messages to the screen.
- bigB** extends the COBOL Area B beyond column 72. The **-bigB** option continues EXEC SQL statements that exceed 72 characters.
- comp89** allows you to include an alternative SQLCA header file.
- EDname** sets a name flag that the UNDEFINE, IFDEF, and IFNDEF preprocessor instructions can use.
- esqlout outfile** changes the name of the preprocessed output file to a name you specify.

-E <i>name</i>	removes an existing name flag specified in the DEFINE preprocessor instruction.
-i <i>check</i>	generates code to check for a null value returned to a host variable that lacks an associated indicator variable, and generates an error when such a case exists.
-I <i>pathname</i>	expands the search range for include directories. The preprocessor also searches the current directory, the <i>pathname</i> , \$INFORMIXDIR/incl , and /usr/include .
-local	keeps the dynamic cursor names and statement identifier names local to the file that defines those names.
-log <i>logfile</i>	logs the error and warning messages in the specified file instead of printing to standard output.
-nowarn	suppresses warning messages from the preprocessor; does not affect error messages.
-t <i>type</i>	specifies the type of compiler being used, as shown in the following examples: MF COBOL/2 -t mf2 RM/COBOL-85 -t rm85
-V	displays the version number of your ESQL/COBOL preprocessor.
-xopen	sets the mode of the ESQL/COBOL program being compiled to X/Open. The default setting for -xopen is normal mode.
= <i>value</i>	lets you assign an INTEGER value to the <i>name</i> flag. For example, -EDMACNAME=62 .

You can use any of these preprocessor options when you preprocess only or when you preprocess, compile, and link.

Checking the Version Number

Use the **-V** argument to learn the version number of your ESQL/COBOL preprocessor, as shown in the following example:

```
esqlcobol -V
```

Including an Alternative SQLCA Header File

To include an alternative SQLCA header file, choose the **-comp89** option. This option allows you to define your own SQLCODE variable without causing a conflict with the Informix-defined SQLCODE variable. For more information on SQLCA and SQLCODE, refer to [Chapter 4, “Error Handling.”](#)

If you use an MF COBOL/2 compiler, the **-comp89** option allows the preprocessor to specify a header file called **sqlca.mf2.alt**, as shown in the following example:

```
*****
*
*   Title: sqlca.mf2.alt
*   Sccsid:   @(#)sqlca.mf2.alt      9.1 1/14/93 13:44:52
*   Description:
*           SQLCA include file for Micro Focus COBOL/2
*****
77  SQLNOTFOUND PIC S9(10) VALUE 100.
01  SQLCA.
    05  IXSQLCODE          PIC S9(9) COMPUTATIONAL-5.
    05  SQLERRM.
        49  SQLERRML      PIC S9(4) COMPUTATIONAL-5.
        49  SQLERRMC      PIC X(70).
    05  SQLERRP            PIC X(8).
    05  SQLERRD            OCCURS 6 TIMES
                          PIC S9(9) COMPUTATIONAL-5.
    05  SQLWARN.
        10  SQLWARN0      PIC X(1).
        10  SQLWARN1      PIC X(1).
        10  SQLWARN2      PIC X(1).
        10  SQLWARN3      PIC X(1).
        10  SQLWARN4      PIC X(1).
        10  SQLWARN5      PIC X(1).
        10  SQLWARN6      PIC X(1).
        10  SQLWARN7      PIC X(1).
```

If you use an RM/COBOL-85 compiler, the **-comp89** option allows the preprocessor to specify a header file called **sqlca.rm.alt**, as shown in the following example:

```
*****
*
* Title: sqlca.rm.alt
* Sccsid:      @(#)sqlca.rm.alt          9.1 1/14/93 13:45:07
* Description:
*           SQLCA include file for Ryan McFarland COBOL
*****
*
77  SQLNOTFOUND PIC S9(10) VALUE 100.
01  SQLCA.
    05  IXSQLCODE          PIC S9(9) COMPUTATIONAL-4.
    05  SQLERRM.
        49  SQLERRML      PIC S9(4) COMPUTATIONAL-1.
        49  SQLERRMC      PIC X(70).
    05  SQLERRP            PIC X(8).
    05  SQLERRD            OCCURS 6 TIMES
                          PIC S9(9) COMPUTATIONAL-4.
    05  SQLWARN.
        10  SQLWARN0      PIC X(1).
        10  SQLWARN1      PIC X(1).
        10  SQLWARN2      PIC X(1).
        10  SQLWARN3      PIC X(1).
        10  SQLWARN4      PIC X(1).
        10  SQLWARN5      PIC X(1).
        10  SQLWARN6      PIC X(1).
        10  SQLWARN7      PIC X(1).
```

The ESQL/COBOL preprocessor knows the correct header file to specify depending on the compiler you use and whether you specify the **-comp89** option.

Checking for ANSI-Standard Syntax

You can check for Informix extensions to ANSI-standard SQL syntax in the following ways when you compile an INFORMIX-ESQL/COBOL program:

- You can set the **DBANSIWARN** environment variable before you proceed. Thereafter, every time you compile or run a program, ESQL/COBOL checks the program automatically for ANSI compatibility. (Refer to the [Informix Guide to SQL: Reference](#) for information on how to set **DBANSIWARN**.)



- Even when you do not set **DBANSIWARN**, you can include the **-ansi** option at compile time whenever you want to check a program for ANSI compatibility. The **-ansi** flag tells the compiler to verify that all SQL statements meet ANSI standards, as shown in the following example:

```
esqlcobol -ansi demo1.eco
```

Warning: *You cannot use the **-ansi** option at run time.*

If you compile with the **-ansi** flag, **esqlcobol** generates warning messages on the screen when you compile a program that either contains Informix extensions to ANSI-standard syntax or uses ANSI reserved words as identifiers. Refer to the [Informix Guide to SQL: Syntax](#) for a list of ANSI reserved words.

If you compile a program with both the **-ansi** and **-xopen** flags, INFORMIX-ESQL/COBOL generates warning messages for both.

Whether or not you compile with the **-ansi** flag, your compiled programs can make run-time checks on Informix extensions to ANSI-standard SQL syntax (even when you do not set the **DBANSIWARN** environment variable). To make a run-time check, use the SQLCA record. INFORMIX-ESQL/COBOL sets the SQLWARN5 OF SQLWARN OF SQLCA field to W when a non ANSI-compliant statement executes. (For more information, refer to [Chapter 4, “Error Handling.”](#))

Checking for Missing Indicator Variables

The **-icheck** option generates code in your program that returns a run-time error when an SQL statement returns a null value to a host variable that lacks an associated indicator variable, as shown in the following example:

```
esqlcobol -icheck demo1.eco
```

If you do not compile using the **-icheck** flag, INFORMIX-ESQL/COBOL does not generate an error in this situation (Refer to [“Indicator Variables and Null Values” on page 1-27](#) in this manual).

Compiling in X/Open Mode

INFORMIX-ESQL/COBOL processes your program using the X/Open set of SQL codes when you include the **-xopen** option on the command line, as shown in the following example:

```
esqlcobol -xopen dynafile.eco
```

When you invoke the **esqlcobol** compiler script using the **-xopen** flag, INFORMIX-ESQL/COBOL uses the X/Open SQL codes when a GET DESCRIPTOR or SET DESCRIPTOR statement executes. Refer to the discussion of these SQL statements in the [Informix Guide to SQL: Syntax](#).

If you include the **-xopen** option on the command line, ESQL/COBOL preprocesses your program using the X/Open set of SQL codes. When you use X/Open SQL in an ESQL/COBOL program, you must recompile all previous programs to make use of the new dynamic SQL statements. You must do this to maintain backward compatibility. When you do not use X/Open SQL in your current ESQL/COBOL programs, ESQL/COBOL does not require you to recompile existing programs.

If you compile a program using both the **-xopen** and **-ansi** flags, ESQL/COBOL generates warning messages for both.

Redirecting Errors and Warnings

ESQL/COBOL automatically sends errors and warnings, generated when you run **esqlcobol**, to standard output. When you want the errors and warnings put in a file, use the **-log** option with the file name, as shown in the following example:

```
esqlcobol -log myerr.err -e mysource.eco
```

Limiting the Scope of Cursor Names and Statement Ids

If you use the **-local** option, dynamic cursor names and statement ids that you declare in a file become local to that file. When you do not use the **-local** option, cursor names and statement ids automatically become global entities, as shown in the following example:

```
esqlcobol -local dynafile.eco
```

If you use the **-local** option, you must recompile the source files every time you rename them. Also, you must make the first 18 characters of the combined cursor names and file names unique. When you mix files compiled with and without the **-local** flag, you get unpredictable results.

Defining and undefining values while preprocessing

You can use the **-ED** and **-EU** options to define or undefine values during preprocessing. Do not put a space between **ED** and the symbol name or between **EU** and the symbol name, as shown in the following example:

```
esqlcobl -EDMACNAME=62 mysource.eco
```

```
esqlcobl -EUMACNAME mysource.eco
```

You can use **-EDname** like you use **DEFINE name**. INFORMIX-ESQL/COBOL processes the **-ED** option before processing the code in your source file.

The **-EU** option is equivalent to using an **UNDEF** statement but has a global effect on the whole file.

Running a Program

The following steps outline the simplest way to preprocess and execute an INFORMIX-ESQL/COBOL program:

1. Write your program. Make sure the source file has the **.eco** extension.
2. Preprocess and compile your program using the following syntax:

```
esqlcobl filename.eco
```

where **esqlcobl** represents the preprocessor command and **filename.eco** represents the name of your source code file.
3. If your program has compilation errors, debug your program and repeat step 2.

4. Run the program as shown in the following examples:

- If you use RM/COBOL-85, use the following syntax:

```
runcobol filename
```

where **runcobol** represents the name of the COBOL run-time program, and *filename* represents the name of your object file.

- If you use MF COBOL/2, use the following syntax:

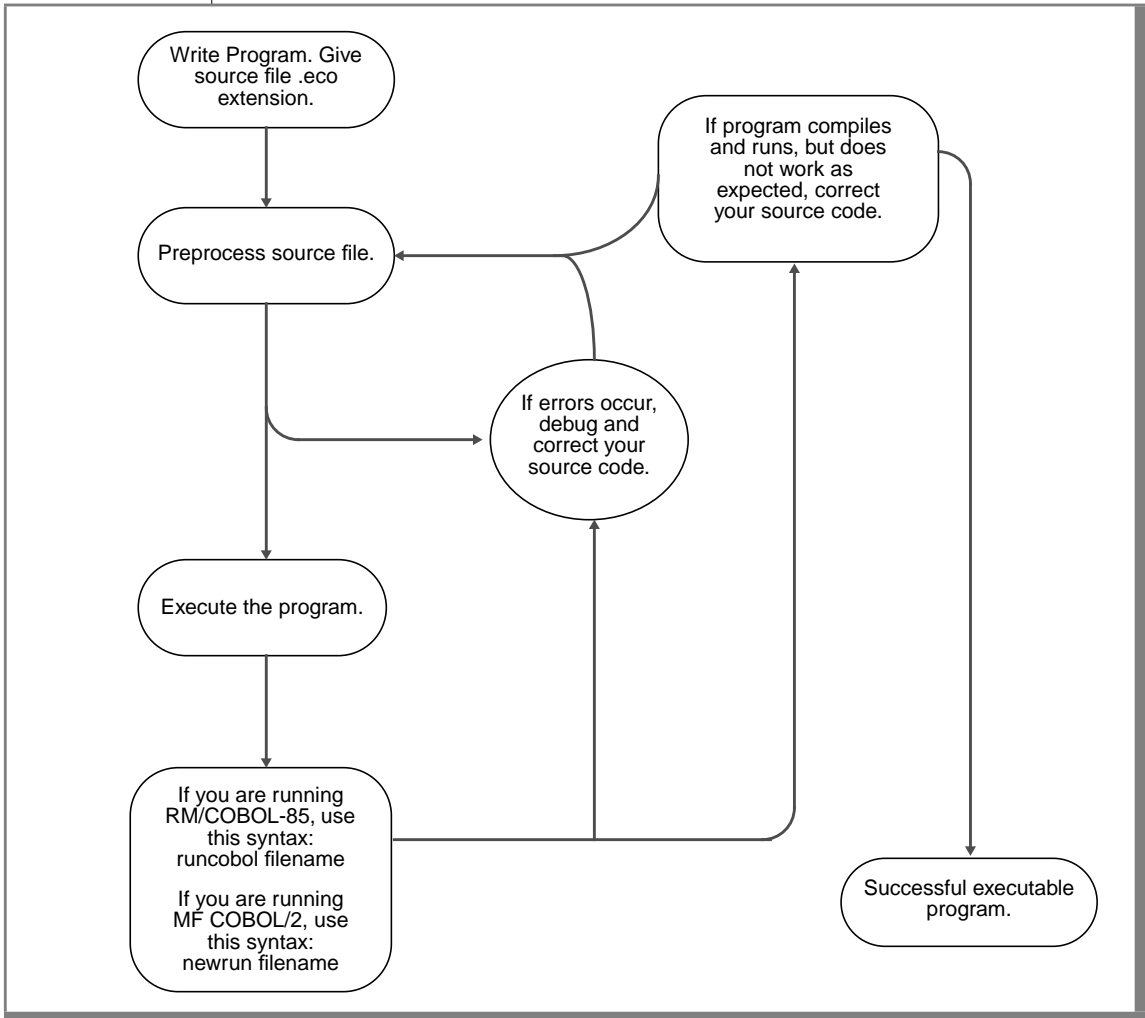
```
newrun filename
```

where **newrun** represents the name of the COBOL run-time program, and *filename* represents the name of your object file.

The preceding examples assume that the name, **runcobol** or **newrun**, represent the name of the COBOL run-time program that you created in the current directory. You can, however, substitute an alternate run-time program name. Refer to [“Creating a COBOL Run-Time Program” on page 1-6](#) in this manual for information on how to create and name a run-time program for your MF COBOL/2 or RM/COBOL-85 compiler.

[Figure 1-12](#) summarizes the process for compiling and running an ESQ/COBOL program.

Figure 1-12
How to Compile and Run an INFORMIX-ESQL/COBOL Program



A Sample INFORMIX-ESQL/COBOL Program

The DEMO1.ECO example beginning on [page 1-46](#) represents one of the sample programs included with your INFORMIX-ESQL/COBOL software. You can create DEMO1.ECO when you respond affirmatively to the `esqlcobdemo7` prompt as discussed in “[Demonstration Database](#)” in the Introduction to this manual. The program works with the MF COBOL/2 and RM/COBOL-85 compilers.

That sample program illustrates most of the concepts that were covered in this chapter, such as a simple use of INCLUDE files, identifiers, host variables, and embedded SQL statements to access and display data from a database. It also shows simple error handling with the SQLSTATE value and the GET DIAGNOSTICS statement.

This ESQL/COBOL demonstration program uses a SELECT statement to declare a cursor. Then ESQL/COBOL opens, fetches, and closes that cursor. At compile time, the program knows and contains all the information needed to run the SELECT statement.

The DEMO1.ECO program reads a subset of the first and last names from the **customer** table in the **stores7** database. Two host variables, :FNAME and :LNAME, hold the data from the **customer** table. The program declares a cursor to manage the information retrieved from the table, and fetches the rows one at a time. The output displays the first and last names of all the customers in the **stores7** database whose last names begin with a letter that has a higher ASCII value than C.

For other examples illustrating dynamic SQL, refer to [Chapter 6](#), “Dynamic Management in INFORMIX-ESQL/COBOL.”

The DEMO1.ECO Program

```
 7  *
 8  *This program, DEMO1, performs a select on the
 9  *customer table of the stores7 demonstration database.
10  *Select and display data (first and last names) by
11  *passing control to several subroutines.
12  *
13  IDENTIFICATION DIVISION.
14  PROGRAM-ID.
15      DEMO1.
16  *
17  ENVIRONMENT DIVISION.
18  CONFIGURATION SECTION.
19  SOURCE-COMPUTER. IFXSUN.
20  OBJECT-COMPUTER. IFXSUN.
21  *
22  DATA DIVISION.
23  WORKING-STORAGE SECTION.
24  *
25  *Declare variables.
26  *
27  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
28  77  FNAME          PIC X(15).
29  77  LNAME          PIC X(20).
30  77  EX-COUNT       PIC S9(9) COMP-5.
31  77  COUNTER        PIC S9(9) VALUE 1 COMP-5.
32  77  MESS-TEXT      PIC X(254).
33  EXEC SQL END DECLARE SECTION END-EXEC.
34  01  WHERE-ERROR    PIC X(72).
35  *
36  PROCEDURE DIVISION.
37  RESIDENT SECTION 1.
38  *
39  *Begin Main routine.  Open a database, declare a cursor,
40  *open the cursor, fetch the cursor, and close the cursor.
41  *
42  MAIN.
43      DISPLAY ' '.
44      DISPLAY ' '.
45      DISPLAY 'DEMO1 SAMPLE ESQL PROGRAM RUNNING.'.
46      DISPLAY '          TEST SIMPLE DECLARE/OPEN/FETCH/LOOP'.
47      DISPLAY ' '.
48
49      PERFORM OPEN-DATABASE.
50
51      PERFORM DECLARE-CURSOR.
52      PERFORM OPEN-CURSOR.
53      PERFORM FETCH-CURSOR
```

```

54         UNTIL SQLSTATE IS EQUAL TO "02000".
55         PERFORM CLOSE-CURSOR.
56         EXEC SQL DISCONNECT CURRENT END-EXEC.
57         DISPLAY 'PROGRAM OVER'.
58     STOP RUN.
59     *
60     *Subroutine to open a database.
61     *
62     OPEN-DATABASE.
63         EXEC SQL CONNECT TO 'stores7' END-EXEC.
64         IF SQLSTATE NOT EQUAL TO "00000"
65             MOVE 'EXCEPTION ON DATABASE STORES7' TO WHERE-ERROR
66             PERFORM ERROR-PROCESS.
67     *
68     *Subroutine to declare a cursor.
69     *
70     DECLARE-CURSOR.
71         EXEC SQL DECLARE DEMOCURSOR CURSOR FOR
72             SELECT FNAME, LNAME
73             INTO :FNAME, :LNAME
74             FROM CUSTOMER
75             WHERE LNAME > 'C'
76         END-EXEC.
77         IF SQLSTATE NOT EQUAL TO "00000"
78             MOVE 'ERROR ON DECLARE CURSOR' TO WHERE-ERROR
79             PERFORM ERROR-PROCESS.
80     *
81     *Subroutine to open a cursor.
82     *
83     OPEN-CURSOR.
84         EXEC SQL OPEN DEMOCURSOR END-EXEC.
85         IF SQLSTATE NOT EQUAL TO "00000"
86             MOVE 'ERROR ON OPEN CURSOR' TO WHERE-ERROR
87             PERFORM ERROR-PROCESS.
88     *
89     *Subroutine to fetch a cursor.  Display data (names).
90     *
91     FETCH-CURSOR.
92         EXEC SQL FETCH DEMOCURSOR END-EXEC.
93         IF SQLSTATE NOT EQUAL TO "00000"
94             AND
95             SQLSTATE NOT EQUAL TO "02000"
96             MOVE 'ERROR DURING FETCH' TO WHERE-ERROR
97             PERFORM ERROR-PROCESS.
98
99         IF SQLSTATE IS EQUAL TO "00000"
100             DISPLAY FNAME, ' ', LNAME.
101     *
102     *Subroutine to close a cursor.

```

```
103 *
104 CLOSE-CURSOR.
105     EXEC SQL CLOSE DEMOCURSOR END-EXEC.
106     IF SQLSTATE NOT EQUAL TO "00000"
107         MOVE 'ERROR ON CLOSE-CURSOR' TO WHERE-ERROR
108         PERFORM ERROR-PROCESS.
109 *
110 *Subroutine to check for exceptions.
111 *
112 ERROR-PROCESS.
113     DISPLAY WHERE-ERROR.
114     DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
115     DISPLAY '*****'.
116     EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
117     PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
118     IF SQLCODE NOT EQUAL TO ZERO
119         STOP RUN.
120 *
121 *Subroutine to print exception messages.
122 *
123 EX-LOOP.
124     EXEC SQL
125         GET DIAGNOSTICS EXCEPTION :COUNTER
126         :MESS-TEXT=MESSAGE_TEXT
127     END-EXEC.
128     DISPLAY 'EXCEPTION ', COUNTER.
129     DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
130     DISPLAY '*****'.
131     ADD 1 TO COUNTER.
132 *
```

Explanation of DEMO1.ECO

The following paragraph-by-paragraph explanation of the DEMO1.ECO source program uses representative COBOL sequence numbering to assist you in locating the program line or concept under discussion.

In this sample program, all paragraph headers begin in Area A. All SQL statements reside within Area B.

Refer to [Chapter 4, “Error Handling.”](#) of this manual and the *Informix Guide to SQL: Syntax* for a description of the SQLSTATE value, and the GET DIAGNOSTICS statement used in this program. The *Informix Guide to SQL: Syntax* describes other SQL statements used in this program.

Lines 1 through 15

The COBOL IDENTIFICATION DIVISION identifies the program, and the ENVIRONMENT DIVISION specifies the computer and any input/output devices that the program uses. This code segment uses the standard COBOL comment indicator, the asterisk (*), in position 7.

```

1  *
2  *This program, DEMO1, performs a select on the
3  *customer table of the stores7 demonstration database.
4  *Select and display data (first and last names) by
5  *passing control to several subroutines.
6  *
7  IDENTIFICATION DIVISION.
8  PROGRAM-ID.
9      DEMO1.
10 *
11 ENVIRONMENT DIVISION.
12 CONFIGURATION SECTION.
13 SOURCE-COMPUTER. IFXSUN.
14 OBJECT-COMPUTER. IFXSUN.
15 *
```

Lines 16 through 29

The DATA DIVISION describes the files, records, and fields that the COBOL program uses.

Host variables represent independent data items. In this example, you declare host variables in the WORKING-STORAGE SECTION as level number 77. (A host variable receives data fetched from a table and supplies data written to a table.) The DEMO1.ECO program defines the FNAME and LNAME host variables as alphanumeric in the PICTURE clauses within the EXEC SQL BEGIN DECLARE SECTION END-EXEC and the EXEC SQL END DECLARE SECTION END-EXEC. The MESS-TEXT host variable receives the contents of the MESSAGE_TEXT field in the GET DIAGNOSTICS statement. The EX-COUNT and COUNTER variables represent conditions that the exception-checking subroutines use.

The alphanumeric nonhost COBOL variable WHERE-ERROR holds an error message string.

```
16 DATA DIVISION.
17   WORKING-STORAGE SECTION.
18   *
19   *Declare variables.
20   *
21   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
22   77 FNAME          PIC X(15).
23   77 LNAME          PIC X(20).
24   77 EX-COUNT       PIC S9(9) COMP-5.
25   77 COUNTER        PIC S9(9) VALUE 1 COMP-5.
26   77 MESS-TEXT      PIC X(254).
27   EXEC SQL END DECLARE SECTION END-EXEC.
28   01 WHERE-ERROR   PIC X(72).
29   *
```

Lines 30 through 53

The PROCEDURE DIVISION contains the actual instructions that the database server performs. It includes the following kinds of statements:

DISPLAY	displays limited output to the screen.
MOVE	moves data from one area of computer storage to another.
PERFORM	branches off to specific paragraphs or subroutines.
PERFORM UNTIL	conditionally transfers control to another subroutine.
STOP	returns control to the operating system.

All the statements in this list reside in the MAIN paragraph, which controls the execution sequence and number of repetitions for each paragraph or subroutine performed.

The DISPLAY lines simply produce screen messages to assure you that the sample program runs correctly and a simple DECLARE/OPEN/FETCH loop executes. A screen message also informs you when the program ends.

The PERFORM statements execute in their listed order in the MAIN paragraph, unless an error occurs. When an error occurs, the program calls the ERROR-PROCESS procedure and program ends without continuing to the next procedure.

Note that the program conditionally performs the FETCH-CURSOR procedure. The loop continues *until* the SQLSTATE variable equals "02000" (In other words, when the program cannot fetch more data). The program automatically initializes the SQLSTATE variable each time an SQL statement executes.

The program ends with the standard STOP RUN statement.

```

30  PROCEDURE DIVISION.
31  RESIDENT SECTION 1.
32  *
33  *Begin Main routine.  Open a database, declare a cursor,
34  *open the cursor, fetch the cursor, and close the cursor.
35  *
36  MAIN.
37      DISPLAY ' '.
38      DISPLAY ' '.
39      DISPLAY 'DEMO1 SAMPLE ESQL PROGRAM RUNNING.'.
40      DISPLAY '          TEST SIMPLE DECLARE/OPEN/FETCH/LOOP'.
41      DISPLAY ' '.
42
43      PERFORM OPEN-DATABASE.
44
45      PERFORM DECLARE-CURSOR.
46      PERFORM OPEN-CURSOR.
47      PERFORM FETCH-CURSOR
48          UNTIL SQLSTATE IS EQUAL TO "02000".
49      PERFORM CLOSE-CURSOR.
50      EXEC SQL DISCONNECT CURRENT END-EXEC.
51      DISPLAY 'PROGRAM OVER'.
52  STOP RUN.
53  *
```

The following pages describe the various PERFORM statements listed in the PROCEDURE division.

Lines 54 through 61

The OPEN-DATABASE subroutine opens the **stores7** database using the embedded SQL statement CONNECT. The words EXEC SQL and END-EXEC contain the embedded SQL statement in the COBOL program.

A conditional IF statement returns the message EXCEPTION ON DATABASE STORES7 to the WHERE-ERROR variable when SQLSTATE does not equal "00000". Such an error can occur when the **stores7** database has not been created before being opened.

If an error occurs during the execution of the OPEN-DATABASE procedure, the database server performs the ERROR-PROCESS procedure.

```
54 *Subroutine to open a database.
55 *
56 OPEN-DATABASE.
57     EXEC SQL CONNECT TO 'stores7' END-EXEC.
58     IF SQLSTATE NOT EQUAL TO "00000"
59         MOVE 'EXCEPTION ON DATABASE STORES7' TO WHERE-ERROR
60         PERFORM ERROR-PROCESS.
61 *
```

Lines 62 through 74

The DECLARE-CURSOR subroutine uses the embedded SQL statement DECLARE to define the cursor DEMOCURSOR for the active set of rows specified in the SELECT statement. The words EXEC SQL and END-EXEC contain the embedded SQL statement in the COBOL program.

The SELECT statement specifies that the :FNAME and :LNAME host variables, that were declared in the DATA DIVISION, to receive the data for the first and last names of customers selected from the **customer** table. In addition, the LNAME must begin with a letter that has a higher ASCII value than C.

An IF statement returns the message ERROR ON DECLARE CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000".

If an error occurs during the execution of the DECLARE-CURSOR procedure, the database server performs the ERROR-PROCESS procedure.

```
62 *Subroutine to declare a cursor.
63 *
64 DECLARE-CURSOR.
65     EXEC SQL DECLARE DEMOCURSOR CURSOR FOR
66         SELECT FNAME, LNAME
67         INTO :FNAME, :LNAME
68         FROM CUSTOMER
69         WHERE LNAME > 'C'
70     END-EXEC.
71     IF SQLSTATE NOT EQUAL TO "00000"
72         MOVE 'ERROR ON DECLARE CURSOR' TO WHERE-ERROR
73         PERFORM ERROR-PROCESS.
74 *
```


Lines 75 through 82

The OPEN-CURSOR subroutine activates the SELECT cursor DEMOCURSOR using the embedded SQL statement OPEN.

An IF statement returns the message ERROR ON OPEN CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000".

If an error occurs during the execution of the OPEN-CURSOR procedure, the database server performs the ERROR-PROCESS procedure.

```

75  *Subroutine to open a cursor.
76  *
77  OPEN-CURSOR.
78      EXEC SQL OPEN DEMOCURSOR END-EXEC.
79      IF SQLSTATE NOT EQUAL TO "00000"
80          MOVE 'ERROR ON OPEN CURSOR' TO WHERE-ERROR
81          PERFORM ERROR-PROCESS.
82  *
```

Lines 83 through 95

The FETCH-CURSOR subroutine uses the embedded SQL statement FETCH to move the cursor DEMOCURSOR to a new row in the active set and to retrieve the row values into memory. DEMOCURSOR selects a row from the **customer** table and puts the data from that row into the host variables :FNAME and :LNAME.

As long as SQLSTATE equals "00000", the program has fetched the data successfully, the FETCH-CURSOR procedure displays the LNAME and FNAME host variables, and the procedure continues.

An IF statement sets the condition where a MOVE statement returns the message ERROR DURING FETCH to the WHERE-ERROR variable when SQLSTATE does not equal "00000" *and* when SQLSTATE does not equal "02000".

If an error occurs during the execution of the FETCH-CURSOR procedure, the database server performs the ERROR-PROCESS procedure.

```
83 *Subroutine to fetch a cursor.  Display data (names).
84 *
85  FETCH-CURSOR.
86      EXEC SQL FETCH DEMOCURSOR END-EXEC.
87      IF SQLSTATE NOT EQUAL TO "00000"
88          AND
89          SQLSTATE NOT EQUAL TO "02000"
90          MOVE 'ERROR DURING FETCH' TO WHERE-ERROR
91          PERFORM ERROR-PROCESS.
92
93      IF SQLSTATE IS EQUAL TO "00000"
94          DISPLAY FNAME, ' ', LNAME.
95 *
```

Lines 96 through 103

The CLOSE-CURSOR subroutine closes the cursor DEMOCURSOR using the embedded SQL statement CLOSE. It disassociates the cursor from the SELECT statement and stops the query process.

A MOVE statement returns the message ERROR ON CLOSE-CURSOR to the WHERE-ERROR variable IF SQLSTATE does not equal "00000".

If an error occurs during the execution of the CLOSE-CURSOR procedure, the database server performs the ERROR-PROCESS procedure.

```
96 *Subroutine to close a cursor.
97 *
98  CLOSE-CURSOR.
99      EXEC SQL CLOSE DEMOCURSOR END-EXEC.
100      IF SQLSTATE NOT EQUAL TO "00000"
101          MOVE 'ERROR ON CLOSE-CURSOR' TO WHERE-ERROR
102          PERFORM ERROR-PROCESS.
103 *
```

Lines 104 through 114

The ERROR-PROCESS subroutine contains the process that counts SQLSTATE exceptions. That subroutine executes whenever an error occurs in one of the other subroutines. The DEMO1.ECO program stops running whenever the ERROR-PROCESS subroutine finds an SQLCODE value not equal to ZERO.

SQLSTATE displays the contents of WHERE-ERROR and indicates the result of executing an SQL statement ("00000", "01000", "02000" or a value greater than "02000"). The GET DIAGNOSTICS NUMBER field contains the count of exceptions associated with the SQLSTATE code. The PERFORM UNTIL statement executes the EX-LOOP subroutine that displays an error message for each exception. When the SQLCODE value does not equal ZERO (success), the program terminates.

```
104 *Subroutine to check for exceptions.
105 *
106 ERROR-PROCESS.
107     DISPLAY WHERE-ERROR.
108     DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
109     DISPLAY '*****'.
110     EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
111     PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
112     IF SQLCODE NOT EQUAL TO ZERO
113         STOP RUN.
114 *
```

Lines 115 through 126

The EX-LOOP subroutine displays the exception number and the error message for each SQLSTATE exception.

```
115 *Subroutine to print exception messages.
116 *
117 EX-LOOP.
118     EXEC SQL
119         GET DIAGNOSTICS EXCEPTION :COUNTER
120         :MESS-TEXT=MESSAGE_TEXT
121     END-EXEC.
122     DISPLAY 'EXCEPTION ', COUNTER.
123     DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
124     DISPLAY '*****'.
125     ADD 1 TO COUNTER.
126 *
```


INFORMIX-ESQL/COBOL Data Types

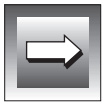
Choosing Data Types for Host Variables	2-4
BINARY or COMP Data Using MF COBOL/2	2-6
Setting the Storage Mode with INFORMIXCOBSTORE.	2-7
Data Conversion.	2-7
Converting CHARACTER Data	2-8
Converting SMALLINT Data	2-8
Converting INTEGER Data.	2-8
Converting FLOAT, SMALLFLOAT, and DECIMAL Data	2-9
Converting DATE Data	2-9
Data Discrepancies During Conversion	2-10
The CHAR Data Type	2-11
CHAR Type Routines	2-13
ECO-DSH.	2-14
ECO-USH	2-17
ECO-GST.	2-20
ECO-SQC.	2-21
The VARCHAR Data Type	2-22
Data Comparison of VARCHAR Values	2-22
Programming with VARCHAR Host Variables	2-23
The TEXT and BYTE Data Types	2-25
Working with Blobs	2-25
Using Blobs with Dynamic SQL	2-27
Using DESCRIBE.	2-28
Using SET DESCRIPTOR	2-28
Using GET DESCRIPTOR.	2-29

Numeric-Formatting Routines	2-31
Formatting Numeric Strings	2-33
ECO-FFL	2-41
ECO-FIN	2-43

This chapter includes information on the SQL and COBOL data types that you can use to manipulate values in your INFORMIX-ESQL/COBOL programs. It covers the following topics:

- The correspondence between the various SQL and COBOL data types.
- How ESQL/COBOL converts data.
- How to use the CHAR (character string) data type and the character string manipulation routines included with the ESQL/COBOL library extension.
- How to use the VARCHAR (variable-length character string) data type in ESQL/COBOL programs.
- How to use TEXT and BYTE data types (blobs) in ESQL/COBOL programs.
- How to format numeric strings and use the numeric-formatting run-time routines included with the ESQL/COBOL library extension.

Refer to the [Informix Guide to SQL: Reference](#) for a detailed description of the SQL data types available in a database. In addition, this chapter and [Chapter 3, “Working with Time Data Types,”](#) describe the syntax and use for all valid ESQL/COBOL data types.



Important: *The NCHAR and NVARCHAR NLS data types match the CHAR and VARCHAR data types, respectively. Information in this chapter regarding the CHAR and VARCHAR data types also applies to the NCHAR and NVARCHAR data types, respectively. For more information on NLS, see the “[Informix Guide to SQL: Reference](#).”*

Choosing Data Types for Host Variables

ESQL/COBOL associates host variables with SQL data types because host variables appear in SQL statements. You must declare a host variable of the appropriate COBOL data type for each column of a table in a database.

Figure 2-1 shows the correspondence between SQL data types and their declaration in ESQL/COBOL.

Figure 2-1
Correspondence Between SQL and COBOL Data Types

SQL Data Type	COBOL Declaration	Notes
BYTE	PIC X(<i>n</i>)	Use the FILE(<i>n</i>) data type to hold the name of the file where you load or store a BYTE column
CHAR(<i>n</i>)	PIC X(10)	
DATE	PIC S9(9) USAGE COMP	Stores Julian values. Use DATE_TYPE (with the COBOL Declaration PIC X(10)), to hold date values in the format <i>mm/dd/yyyy</i>
DECIMAL(<i>p,n</i>)	PIC S9(<i>m</i>)V9(<i>n</i>) USAGE COMP-3	<i>m = p - n</i>
INT	PIC S9(9) USAGE COMP	
MONEY(<i>p,n</i>)	PIC S9(<i>m</i>)V9(<i>n</i>) USAGE COMP-3	<i>m = p - n</i>
NCHAR(<i>n</i>)	PIC X(10)	
NVARCHAR(<i>n</i>)	PIC X(<i>n</i>)	

(1 of 2)

SQL Data Type	COBOL Declaration	Notes
SMALLINT	PIC S9(4) USAGE COMP	
TEXT	PIC X(<i>n</i>)	Use the FILE(<i>n</i>) data type to hold the name of the file where you load or store a TEXT column
VARCHAR(<i>n</i>)	PIC X(<i>n</i>)	
* For MF COBOL/2, substitute COMP-5 for COMP; for RM/COBOL-85, substitute COMP-1 for COMP.		

(2 of 2)

You must declare ESQL/COBOL host variables in the EXEC SQL BEGIN DECLARE SECTION END-EXEC portion of a program. These host variables can use PICTURE clauses containing the following characters:

- X, S, 9, and V
- The repetition expression (*n*)

PICTURE clauses for ESQL/COBOL host variables do *not* allow you to use the following characters:

- A and P
- Editing characters such as Z, /, +, -, ., and ,

You can use any USAGE clause when you define a host variable. (Refer to [“Using Host Variables in SQL Statements” on page 1-21](#) for information on declaring host variables.)

The ESQL/COBOL preprocessor generates the correct COBOL data type in the .cob file. This ensures that ESQL/COBOL allocates consistently the correct amount of memory for the host variables. (Refer to [“Compiling INFORMIX-ESQL/COBOL Programs” on page 1-32.](#))

BINARY or COMP Data Using MF COBOL/2

INFORMIX-ESQL/COBOL supports only those data types valid for the particular implementation of COBOL. The ESQL/COBOL preprocessor reports as illegal any variables defined with a data type that COBOL does not recognize.

INFORMIX-ESQL/COBOL using MF COBOL/2 can report additional errors for BINARY and COMPUTATIONAL (COMP) data types because of limits on the storage sizes of these data types.

The size (maximum number of digits) specified in the PICTURE clause provides the basis for the number of bytes needed to store BINARY or COMPUTATIONAL data. When determining the number of bytes needed to store BINARY and COMPUTATIONAL data, MF COBOL/2 compilers allocate storage based on whether you specify *byte* storage or *word* storage.

Figure 2-2 shows the storage allocation for MF COBOL/2 compilers.

Figure 2-2
Storage Allocation for MF COBOL/2 Compilers

Number of Digits in PICTURE		Variable Size (in bytes)	
Signed	Unsigned	Byte Storage	Word Storage
1-2	1-2	1	2
3-4	3-4	2	2
5-6	5-7	3	4
7-9	8-9	4	4
10-11	10-12	5	8
12-14	13-14	6	8
15-16	15-16	7	8
17-18	17-18	8	8

The COBOL numeric data types must match internal data types to properly interface to the INFORMIX-ESQL/COBOL libraries. To make sure that host variables types match each other, the ESQL/COBOL preprocessor reports an error when the resulting variable size does not equal 2 or 4 bytes.

In addition, INFORMIX-ESQL/COBOL limits byte and word storage to the following specific PICTURE clause sizes:

- When you use byte storage, only 3, 4, 7, 8, and 9 represent legal PIC sizes.
- When you use word storage, PIC sizes can range from 1 to 9, inclusive.

Setting the Storage Mode with INFORMIXCOBSTORE

You can use the environment variable **INFORMIXCOBSTORE** to indicate to INFORMIX-ESQL/COBOL the type of storage mode to use during compilation in the MF COBOL/2 environment. You can use only word and byte as legal values for **INFORMIXCOBSTORE**. When you leave **INFORMIXCOBSTORE** undefined, then storage mode defaults to byte. The [Informix Guide to SQL: Reference](#) discusses **INFORMIXCOBSTORE** and other environment variables in greater detail.

Data Conversion

The relationship of the SQL data types to the PICTURE clause used in the declaration of COBOL host variables determines the conversion of data stored in an SQL database and COBOL host variables.

If you enter data into an SQL database exclusively through an ESQL/COBOL program or from a collection of data that another COBOL program generates, you can choose both SQL and COBOL data types so that data conversion in both directions can proceed without loss of accuracy and without unacceptable rounding or truncation of data.

Warning: Exercise caution when you incorporate data that another program prepares. You can encounter data conversion problems when the other program uses data types that do not correspond to INFORMIX-ESQL/COBOL data types.



Converting CHARACTER Data

The SQL data type `CHAR(n)` corresponds exactly to the COBOL PICTURE `X(n)`, and vice versa.

For CHARACTER data (and some INTEGER type data) you can assign SQL and COBOL data types in a way that guarantees that data passes in both directions without any loss of accuracy or other conversion failures.



Important: When you use NLS, the `NCHAR` and `NVARCHAR` NLS data types correspond to the `CHAR` and `VARCHAR` data types, respectively. The preceding NLS data types allow you to use `CHAR` and `VARCHAR` data in an NLS environment without causing data conversion problems. For more information on NLS, see the [“Informix Guide to SQL: Reference.”](#)

Converting SMALLINT Data

You can insert a COBOL PICTURE `S9(4)` value into an SQL `SMALLINT` column. However, a problem can occur when you convert in the other direction because the largest `SMALLINT` equals 32,767. For example, a conversion failure occurs when you try to pass the number 30,000 from an SQL `SMALLINT` column to a COBOL variable with PICTURE `S9(4)`.

Converting INTEGER Data

You can insert a COBOL PICTURE `S9(9)` into an SQL `INTEGER` column. However, a problem can occur when you convert in the other direction with `INTEGER` and PICTURE `S(9)` because the largest `INTEGER` equals 2,147,483,647.

If your `INTEGER` data probably exists within `INTEGER` data limits, make sure you provide adequate storage size in both directions. To provide storage size, choose a data type with a larger range.

If you think integers that exceed the size of the largest `INTEGER` can occur in your program, use the `DECIMAL(s)` data type where *s* = 10.

Converting FLOAT, SMALLFLOAT, and DECIMAL Data

The conversion of data with fractional parts is more complex than the conversion of character and integer data. COBOL numeric data spans from the decimal numbers 10^{-18} [PIC SV9(18)] to 10^{+18} [PIC S9(18)], whereas FLOAT, SMALLFLOAT, and DECIMAL exceed that range. When you create all the data using COBOL, the preceding conversion succeeds.

COBOL automatically stores values in base 10, and stores SMALLFLOAT and FLOAT in base 2. Inevitable rounding occurs with conversion in either direction. Informix, therefore, recommends that you use DECIMAL types for database columns when possible.

When you convert FLOAT, SMALLFLOAT, or DECIMAL from COBOL to SQL, enter the data type PIC S9(*m*)V9(*n*) into a database column of one of these types:

- DECIMAL(*p,s*), where $p = m+n$
- FLOAT, where $m+n = 14$
- SMALLFLOAT, where $m+n = 7$

When you convert FLOAT, SMALLFLOAT, or DECIMAL from SQL to COBOL, use PIC S9(*m*)V9(*n*), where *m* represents the exponent to *base 10* of the largest likely absolute value and *-n* represents the exponent to *base 10* of the smallest likely value. To preserve precision, make *n* as large as possible, consistent with your storage capacity.

Converting DATE Data

ESQL/COBOL stores the DATE data type internally as a 4-byte integer. You can define a corresponding host variable in one of the following ways:

- If you use that variable strictly within SQL statements, you can define the corresponding host variable as the 4-byte integer [PIC S9(9)].
- You can use the special DATE_TYPE clause [PIC X(10)] to put the host variable in a user-readable *mm/dd/yyyy* form. (Refer to “[DATE Type Routines](#)” on page 3-3.)

Data Discrepancies During Conversion

When a discrepancy exists between the data type of a database value and the data type of the host variable, or between the data types of two columns, ESQL/COBOL tries to convert one into the other. For example, ESQL/COBOL converts a CHAR data type into a number data type when the CHAR variable represents a number.

When comparing a CHAR value and a number value, ESQL/COBOL converts the CHAR value to a number value. To convert a number data type to a CHAR data type, ESQL/COBOL creates a string.

If ESQL/COBOL cannot make a conversion, because of an unmeaningful conversion or because the converted value exceeds the size of the receiving variable, ESQL/COBOL returns values as described in Figure 2-3. In that figure, **Num** represents a number data type, and **Char** represents a character data type.

Figure 2-3
Conversion Discrepancies and Results for Number and Character Types

Conversion	Discrepancy	Result
Char to Char	Does not fit	ESQL/COBOL truncates the string, and sets the indicator variable to the length of the SQL column. (Note that even the truncation of trailing blanks triggers these results.)
Num to Char	Does not fit	ESQL/COBOL fills the string with asterisks, and sets the indicator variable to a positive integer.
Char to Num	Not a number	Undefined number. ESQL/COBOL makes SQLCODE OF SQLCA negative and sets the indicator variable to nonzero.
Char to Num	Overflow	Undefined number. ESQL/COBOL makes SQLCODE OF SQLCA negative.
Num to Num	Overflow	Undefined number. ESQL/COBOL makes SQLCODE OF SQLCA negative.



Important: The CHAR data type information also applies to the NCHAR NLS data type. For more information on NLS, see the “[Informix Guide to SQL: Reference](#).”

In Figure 2-3, the phrase *does not fit* means that the sending variable exceeds the size of the receiving character variable or column. When the fractional part of a number does not fit in a character variable, ESQL/COBOL rounds that number. Asterisks (*) appear only when the integer part does not fit.

The term *overflow* describes a situation where the sending number (or character representation of a number) exceeds in absolute magnitude the largest value that the data type of the receiving variable permits.

Another conversion problem not listed in Figure 2-3 occurs when the smallest COBOL number exceeds the size of an SQL number, that is called *underflow*. For example, this problem occurs when you place a positive number less than 10^{-5} into a COBOL variable with the data type PIC SV9(5). In the case of underflow, ESQL/COBOL stores the value zero and sets no error or warning flags.

When conversion problems occur, ESQL/COBOL provides warning and error trapping. In each case listed in Figure 2-3, ESQL/COBOL sets SQLWARN1 OF SQLWARN OF SQLCA to W. When a right truncation occurs on a data string, ESQL/COBOL sets the SQLSTATE value to a warning value of 01004.

Chapter 4, “Error Handling,” describes the SQLCA record. Refer also to “Indicator Variables and Null Values” on page 1-27 for a discussion of how you can use indicator variables to detect conversion errors.

The CHAR Data Type

The CHAR data type (also known as CHARACTER) stores any string of *printable* letters, numbers, and symbols. ESQL/COBOL stores a CHAR value in its row in the tblspace. When ESQL/COBOL stores a CHAR value, ESQL/COBOL uses spaces to pad that value to the size of its column.

The NCHAR data type (short for national character) also stores any string of printable letters, numbers, and symbols when NLS has been enabled. ESQL/COBOL treats NCHAR the same as a CHAR except that the current release does not support comparisons between NCHAR and CHAR values.

For more information on NLS and the CHAR and NCHAR data types, refer to the [Informix Guide to SQL: Reference](#).

Figure 2-4 shows the corresponding COBOL data type and COBOL description for the arguments used in the CHAR manipulation routines discussed in this chapter.

Figure 2-4
COBOL Types and Descriptions for CHAR Manipulation Routines

Argument	COBOL Type	COBOL Description
<i>S</i>	CHARACTER	PIC X(LENGTH)
<i>S-LEN</i>	INTEGER	PIC S9(?) COMP

ESQL/COBOL implements the PIC S9(?) shown in the COBOL Description column as PIC S9(9) for a 4-byte INTEGER or PIC S9(4) for a 2-byte INTEGER.

Figure 2-4 also lists the word COMP located in the COBOL Description column. To interpret the word COMP, refer to Figure 2-5 for a listing of COMP equivalents for the types of COBOL compilers that ESQL/COBOL supports.

Figure 2-5
COMP Equivalents for Supported COBOL Compilers

COMP Equivalent	Type of COBOL
COMP-5	MF COBOL/2
COMP-1	RM/COBOL-85

CHAR Type Routines

Figure 2-6 shows two CHAR string-manipulation routines and two related error-checking routines included in the libraries distributed with INFORMIX-ESQL/COBOL.

Figure 2-6
Descriptions of CHAR Type Routines

Routine Name	What It does
ECO-DSH	Converts a character string to lowercase
ECO-USH	Converts a character string to uppercase
ECO-GST	Checks ECO-DSH, ECO-USH, and ECO-DAT for errors
ECO-SQC	Checks ECO-DSH, ECO-USH, and ECO-DAT for errors

Use the ECO-DSH and ECO-USH routines in your COBOL programs to manipulate CHAR data types. When you use the **esqlcobol** compiler shell script, ESQL/COBOL links the run-time routines automatically.

Some routines (for example, ECO-DSH and ECO-USH in this chapter, and ECO-DAT in [Chapter 3, “Working with Time Data Types”](#)) lack a status parameter. Use the ECO-GST and ECO-SQC routines to check for errors returned those types of routines.

The following four sections provide detailed descriptions of the routines listed in Figure 2-6.

ECO-DSH

Purpose

Use ECO-DSH to *downshift*, that is, convert all the characters within a character string to lowercase characters.

Syntax

```
CALL ECO-DSH USING S, S-LEN.
```

S the character string that you provide

S-LEN the length that you specify for *S*

Usage

ECO-DSH lacks a *STATUS* parameter. Instead, this routine sets the internal variable SQLCODE OF SQLCA. To check the SQLCA variable, call the ECO-GST or ECO-SQC routine. Refer to pages [2-20](#) and [2-21](#), respectively, for descriptions of those routines.

After you call the ECO-DSH routine, you can test SQLCODE with the ECO-GST or ECO-SQC routines to check SQLCODE for an error code (zero means no error). Refer to [Chapter 4, “Error Handling,”](#) for a discussion of error handling and the SQLCA record.

GLS

The ECO-DSH routine can handle multibyte characters in its strings. In addition, this routine supports locale-specific definitions of uppercase and lowercase characters. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Example

The following code fragment shows how to use ECO-DSH to convert an uppercase string to a lowercase string. Note that the data type of TEXTLEN1 depends on the type of compiler used. The ECO-DSH routine converts the uppercase string contained in the variable TEXT1 to a lowercase string. Then, the ECO-GST routine checks for any errors.

```

1 *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODSH.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
12 77 TEXT1          PIC X(17) VALUE "123ABCDEFHIJK".
13 77 TEXTLEN1       PIC S9(9) USAGE COMP-5 VALUE 17.
14 EXEC SQL END DECLARE SECTION END-EXEC.
15 *
16 PROCEDURE DIVISION.
17 RESIDENT SECTION 1.
18 *
19 *Begin Main routine. Display the original string.
20 *Convert the string to lowercase using the ECO-DSH
21 *routine. Display the converted string.
22 *
23 MAIN.
24     DISPLAY 'TEXT BEFORE DOWNSHIFT IS: ', TEXT1.
25     CALL ECO-DSH USING TEXT1, TEXTLEN1.
26     DISPLAY 'TEXT AFTER DOWNSHIFT IS: ', TEXT1.
27     CALL ECO-GST USING SQLCA.
28     DISPLAY 'SQLCODE VALUE IS: ', SQLCODE.
29 STOP RUN.
30 *
```

Example Output

The output for the preceding code fragment displays the original character string, the converted character string in lowercase letters, and the SQLCODE value.

```
TEXT BEFORE DOWNSHIFT IS: 123ABCDEFGHIJK  
TEXT AFTER DOWNSHIFT IS: 123abcdefghijk  
SQLCODE VALUE IS: +0000000000
```

ECO-USH

Purpose

Use ECO-USH to *upshift*, that is, to convert all the characters within a character string to uppercase characters.

Syntax

```
CALL ECO-USH USING S, S-LEN.
```

S the character string that you provide
S-LEN the length that you specify for *S*

Usage

ECO-USH lacks a *STATUS* parameter. Instead, ECO-USH sets the internal variable `SQLCODE OF SQLCA`. To check the `SQLCA` variable, call the ECO-GST or ECO-SQC routine. Refer to pages [2-20](#) and [2-21](#), respectively, for descriptions of those routines.

After you call the ECO-DAT routine, you can test `SQLCODE` with the ECO-GST or ECO-SQC routine to check `SQLCODE` for an error code (zero indicates no error). Refer to [Chapter 4, “Error Handling,”](#) for a discussion of error handling and the `SQLCA` record.

GLS

The ECO-USH routine can handle multibyte characters in its strings. In addition, this routine supports locale-specific definitions of uppercase and lowercase characters. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Example

The following code fragment shows how to use ECO-USH to convert a lowercase string to an uppercase string. Note that the data type of TEXTLEN1 depends on the type of compiler used.

```

1 *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOUSH.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 *Declare variables.
12 *
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 TEXT1          PIC X(17) VALUE "123abcdefghijkl".
15 77 TEXTLEN1       PIC S9(9) USAGE COMP-5 VALUE 17.
16 EXEC SQL END DECLARE SECTION END-EXEC.
17 *
18 PROCEDURE DIVISION.
19 RESIDENT SECTION 1.
20 *
21 *Begin Main routine. Display the string. Call the
22 *ECO-USH routine to convert the string to
23 *uppercase characters. Display the converted
24 *string.
25 *
26 MAIN.
27     DISPLAY 'Text before upshift is: ', TEXT1.
28     CALL ECO-USH USING TEXT1, TEXTLEN1.
29     DISPLAY 'Text after upshift is: ', TEXT1.
30     CALL ECO-GST USING SQLCA.
31     DISPLAY 'SQLCODE value is: ', SQLCODE.
32 STOP RUN.
33 *
```

Example Output

The output for the preceding code fragment displays the original character string, the converted character string in uppercase letters, and the SQLCODE value.

```
TEXT BEFORE UPSHIFT IS: 123abcdefghijkl  
TEXT AFTER UPSHIFT IS: 123ABCDEFGHIJK  
SQLCODE VALUE IS: +0000000000
```

ECO-GST

Purpose

The ECO-DAT, ECO-DSH, and ECO-USH ESQL/COBOL routines all lack a *STATUS* parameter. Use ECO-GST to check the SQLCODE OF SQLCA when you call those routines.

Syntax

```
CALL ECO-GST USING SQLCA.  
SQLCA           the SQLCA record
```

Usage

SQLCA represents the SQLCA record defined in the INCLUDE SQLCA statement. Refer to the discussion of INCLUDE statements on page [1-30](#) for more information. [Chapter 4, “Error Handling,”](#) discusses the structure and use of the SQLCA record in ESQL/COBOL.



Tip: *ECO-SQC exceeds the functionality of ECO-GST.*

ECO-SQC

Purpose

The ECO-DAT, ECO-DSH, and ECO-USH ESQL/COBOL routines all lack a *STATUS* parameter. Use ECO-SQC to check the SQLCODE OF SQLCA and the SQLWARN OF SQLCA codes when you call those routines.

Syntax

```
CALL ECO-SQC USING SQLCA, SQLCODETMP, SQLWARNTMP.
```

SQLCA	the SQLCA record
SQLCODETMP	the SQL result code, SQLCODE OF SQLCA (INTEGER)
SQLWARNTMP	the SQL warning code, SQLWARN0 OF SQLWARN OF SQLCA (INTEGER)

Usage

The SQLCA parameter represents the SQLCA record defined in the INCLUDE SQLCA statement. The preprocessor includes that record automatically in the COBOL generated file. Refer to the discussion of INCLUDE statements on page [1-30](#) for more information.

The SQLCODETMP and SQLWARNTMP parameters represent temporary variables that the preprocessor generates. ESQL/COBOL does *not* require you to declare those variables.

The SQLCODE OF SQLCA indicates the result of executing an SQL statement, and SQLWARN OF SQLCA signals various warning conditions. [Chapter 4, “Error Handling,”](#) discusses the structure and use of the SQLCA record in ESQL/COBOL.

ECO-SQC exceeds the functionality of ECO-GST.

The VARCHAR Data Type

The term VARCHAR means *variable character*. You can use a VARCHAR to define the data type of a column in ESQL/COBOL programs only when you use an INFORMIX-OnLine Dynamic Server.

The VARCHAR data type stores a character string of varying length that can range in size from 1 byte to 255 bytes. In ESQL/COBOL, you use VARCHAR values much like CHAR values. You can create, fetch, compare, index, subscript, and display a VARCHAR value just like a CHAR value. Like a CHAR value, ESQL/COBOL stores the VARCHAR value in its row in the tblspace.

The term NVARCHAR means native variable character. You can use an NVARCHAR to define the data type of a column in INFORMIX-ESQL/COBOL programs only when you use an INFORMIX-OnLine Dynamic Server and you enable NLS. NVARCHAR allows for foreign characters. You treat NVARCHAR the same as VARCHAR.

For more information on the VARCHAR and NVARCHAR data types, refer to the [Informix Guide to SQL: Reference](#). For more information on NLS, see the Informix 7.1 documentation.



Important: In ESQL/COBOL Version 7.2, no run-time routines or macros, designed specifically to manipulate VARCHAR data types, exist.

Data Comparison of VARCHAR Values

ESQL/COBOL compares VARCHAR values to other VARCHAR values and to CHARACTER values in the same way that ESQL/COBOL compares CHARACTER values. This means that ESQL/COBOL pads the shorter value on the right with spaces until the value lengths match. Then, ESQL/COBOL compares those values for the full length. The same holds true for NVARCHAR values when you enable NLS, except that ESQL/COBOL Version 7.2 does not support comparisons between NVARCHAR and VARCHAR values. For more information on NLS, see the Informix 7.1 documentation.

[Figure 2-7](#) lists some examples (the plus sign [+] represents a space) of VARCHAR and NVARCHAR values:

Figure 2-7
Examples of VARCHAR and NVARCHAR Values

Type	Length	Data	Type	Length	Data	Result
VARCHAR	2	CA	VARCHAR	2	CA	equal
VARCHAR	3	AB+	VARCHAR	2	AB	equal
VARCHAR	4	abcd	CHARACTER	5	abcd+	equal
VARCHAR	3	OH+	CHARACTER	2	OH	equal
NVARCHAR	2	CA	NVARCHAR	2	CA	equal
NVARCHAR	3	AB+	NVARCHAR	2	AB	equal

Programming with VARCHAR Host Variables

ESQL/COBOL strips the trailing spaces in VARCHAR host variables (and NVARCHAR host variables when you enable NLS) when you store the value in the database. When you retrieve a value from a VARCHAR column in the database, ESQL/COBOL pads that value with spaces to the declared length of the VARCHAR host variable. For more information on NLS, see the [Informix Guide to SQL: Reference](#).

Figure 2-8 shows two examples of the contents of a VARCHAR host variable and their associated values as stored in the database (a plus sign [+] represents a space).

Figure 2-8
Examples of VARCHAR Host Variable Contents and Values

Host Variable		Database Column	
Type	Data	Type	Data
VARCHAR	Cincinnati+	VARCHAR	Cincinnati
VARCHAR	Cincinnati	VARCHAR	Cincinnati

Figure 2-9 shows what happens when you select a VARCHAR value from the database into a VARCHAR host variable (a plus sign [+] represents a space).

Figure 2-9
*Example of Result Caused When You Select a
VARCHAR Value from Database into VARCHAR Host Variable*

Database Column		Host Variable		
Type	Data	Type	Length	Data
VARCHAR	Cincinnati	VARCHAR	15	Cincinnati+++++

Use the following syntax to declare a host variable for a VARCHAR in an ESQL/COBOL program:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
  
*... (other SQL declarations)  
  
77 VC VARCHAR(n).  
  
*... (other SQL declarations)  
  
EXEC SQL END DECLARE SECTION END-EXEC.
```

Replace *VC* with the name of the host variable. Replace *n* with the *max-size* of the VARCHAR as you specified in your CREATE TABLE statement.

Then, ESQL/COBOL generates the following COBOL code:

```
77 VC PIC X(n).
```

For example, in the procedure division of your program, imagine that you specified the following column definition:

```
EXEC SQL CREATE TABLE EMPLOYEE (  
    LNAME      CHAR(20),  
    FNAME      CHAR(20),  
    START_DATE DATE,  
    HISTORY     VARCHAR(200, 50) )  
END-EXEC.
```

This column definition requires that you declare a host variable in the data division, as shown in the following code fragment:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

77 H-HISTORY VARCHAR(200).

EXEC SQL END DECLARE SECTION END-EXEC.
```

The TEXT and BYTE Data Types

This section describes how to work with blobs (binary large objects) in ESQL/COBOL. A blob represents a data object that theoretically has no maximum size. In addition, you must specify the data type of a blob as TEXT or BYTE.

The TEXT data type stores any kind of text data. The BYTE data type stores any kind of binary data in an undifferentiated byte stream. You can use the TEXT and BYTE data types in ESQL/COBOL to define the data type of a column in database applications only when you use INFORMIX-OnLine Dynamic Server.

For more information on the TEXT and BYTE data types, refer to the [Informix Guide to SQL: Reference](#). Refer also to the [INFORMIX-OnLine Dynamic Server Administrator's Guide](#) for information on locks and how ESQL/COBOL stores blobs.



Important: *Informix does not include any Informix run-time routines, that manipulate only TEXT or BYTE data types, with INFORMIX-ESQL/COBOL Version 7.2.*

Working with Blobs

You can insert data into TEXT and BYTE columns from a FILE host data type in ESQL/COBOL. You cannot use a quoted text string, number, or any other actual value to insert or update blob columns.

In ESQL/COBOL, you can access blobs through a special type called FILE. With standard SQL statements, you can select data as shown in the following example:

```
SELECT XBLOB INTO :FILEVAR FROM TAB
```

To use the TEXT and BYTE data types with ESQL/COBOL simply specify a file used to store the blob. You need to declare only the name of the host variable to use the blob data type.

The following code fragment shows how to declare a host variable for a column that uses the TEXT or BYTE data type in ESQL/COBOL:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
  
* . . . (other SQL declarations)  
  
77 BLOB-VAR FILE(n) VALUE 'filename'.  
  
* . . . (other SQL declarations)  
  
EXEC SQL END DECLARE SECTION END-EXEC.
```

Replace *BLOB-VAR* with the name of the host variable, *n* with the number of letters in the name of the file, and *filename* with the name that you specify for the file.

The following example shows the COBOL code that ESQL/COBOL generates:

```
77 BLOB-VAR PIC X(n) VALUE 'filename'.
```

For example, in the procedure division of your program, imagine that you specify the following TEXT column definition:

```
EXEC SQL CREATE TABLE EMPLOYEE (  
    LNAME          CHAR(20),  
    FNAME          CHAR(20),  
    START_DATE     DATE,  
    HISTORY         VARCHAR(200, 50),  
    RESUME         TEXT )  
END-EXEC.
```

This column definition requires you to declare a host variable in the data division as shown in the following code fragment:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
  
77 H-RESUME FILE(8) VALUE 'res-file'.  
  
EXEC SQL END DECLARE SECTION END-EXEC.
```

Using Blobs with Dynamic SQL

You can use binary large objects (blobs) with SQL descriptors in X/Open dynamic SQL. To access blobs correctly in ESQL/COBOL, you must perform the following steps before you execute a DESCRIBE statement:

1. Initialize the TYPE field of the system-descriptor area to 116.
2. Initialize the LENGTH field of the system-descriptor area to the length of the FILE variable.
3. Initialize the DATA field of the system-descriptor area using the FILE variable.



Warning: *You cannot rely on the DESCRIBE statement to set up the returned values correctly.*

The following example shows how to use an SQL descriptor to retrieve a BYTE data column into a file named **blob_output**. The example assumes that you name the table **tab** and you name the blob field **col**.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
77 FILENAME FILE(12) VALUE 'blob_output'.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SET DESCRIPTOR 'desc' VALUE 1, TYPE = 116,
      DATA = :FILENAME, LENGTH = 12
END-EXEC.
...
EXEC SQL DECLARE x CURSOR FOR SELECT COL FROM TAB END-EXEC.
...
EXEC SQL FETCH x USING SQL DESCRIPTOR 'desc' END-EXEC.
...
```

With dynamic SQL, you can manipulate the system descriptors to access blob data. The examples shown in [“Using SET DESCRIPTOR” on page 2-28](#) and [“Using GET DESCRIPTOR” on page 2-29](#) show how to use the FILE variable with X/Open dynamic SQL statements.

Using DESCRIBE

After describing an ESQL/COBOL statement that involves a blob column, you set the system-descriptor area entry fields for the column to the values shown in Figure 2-10.

Figure 2-10

Example of System-Descriptor Area Entries for Blob Column

Value	Description
TYPE	SQLTEXT or SQLBYTE
LENGTH	56 (length of an internal blob structure)
NAME	name of the column
NULLABLE	set when column allows nulls

Make sure you leave the contents of all other fields undefined.

Using SET DESCRIPTOR

Before you can use the blob data, you must initialize that data with a variable of type FILE. You can choose a shorter length, but you must change the TYPE field of the system descriptor area to a value of 116. For example, the following statement specifies that a file named **outputfile** can access the blob:

```
MOVE 'outputfile' TO FILEVAR.
SET DESCRIPTOR 'desc' VALUE 3 DATA = :FILEVAR, LENGTH = 20,
    TYPE = 116.
```


Using GET DESCRIPTOR

You can use the following GET DESCRIPTOR statement on a blob **sqlvar** to find out the name of the FILE that you already set:

```
GET DESCRIPTOR 'desc' VALUE 3 :FILEVAR2 = DATA
```

If you specified **outfile** as the file you set, FILEVAR2 contains the name **outfile**, provided that the variable equals or exceeds the size of the file name.

Assuming that you set the descriptor properly, the following FETCH statement puts the contents of the blob data in the specified file **outfile**:

```
FETCH AC USING SQL DESCRIPTOR 'desc'
```

The following example program, COBLOB, works with TEXT data. COBLOB creates a test database, creates a database table that contains a blob column, and stores the text of the source code in that table. You can verify that this program works using DIFF to compare the contents of **coblob.eco** and **rettext01**. This example works only with MF COBOL/2.

```
1 *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      COBLOB.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 INPUT-OUTPUT SECTION.
12 FILE-CONTROL.
13 DATA DIVISION.
14 FILE SECTION.
15 *
16 *Declare variables.
17 *
18 WORKING-STORAGE SECTION.
19 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
20 01 HTEXT1    FILE(13).
21 01 TRET1     FILE(9).
22 EXEC SQL END DECLARE SECTION END-EXEC.
23 PROCEDURE DIVISION.
24 *
```

```
25  *Begin Main routine. Create a database. Create a
26  *table with a column. Move the source code of this
27  *program into a file host variable. Insert the
28  *file host variable into the table column. Move
29  *an empty external file into another host variable.
30  *SELECT the column containing the blob into the
31  *file host variable containing the blank external
32  *file. CLOSE and DROP the database. Inspect the
33  *external file "retttext01" after the program ends
34  *to verify that it contains blob data.
35  *
36  MAIN.
37      EXEC SQL
38          CONNECT TO DEFAULT
39      END-EXEC.
40      EXEC SQL CREATE DATABASE testname END-EXEC.
41      EXEC SQL
42          CREATE TABLE tab1 (column1 text)
43      END-EXEC.
44      MOVE "coblob.eco" TO HTEXT1.
45      EXEC SQL
46          INSERT INTO tab1 values (
47              :HTEXT1)
48      END-EXEC.
49      MOVE "retttext01" TO TRET1.
50      EXEC SQL
51          SELECT column1 INTO
52              :TRET1
53          FROM tab1
54      END-EXEC.
55      EXEC SQL
56          CLOSE DATABASE
57      END-EXEC.
58      EXEC SQL
59          DROP DATABASE TESTNAME
60      END-EXEC.
61      EXEC SQL DISCONNECT ALL END-EXEC.
62  STOP RUN.
63  *
```

Numeric-Formatting Routines

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

Figure 2-11 shows the numeric-formatting routines included in the libraries distributed with INFORMIX-ESQL/COBOL. This section alphabetically lists and describes those routines.

Figure 2-11
Descriptions of Numeric-Formatting Routines

Routine Name	What It Does
ECO-FFL	Returns a character string for a floating-point value
ECO-FIN	Returns a character string for an INTEGER value

In ESQL/COBOL, the routines listed in the preceding table return a character-string representation of a given value for a specified format. The implementation defines an INTEGER. In other words, INTEGER represents 4-byte when supported, otherwise INTEGER represents 2-byte.

Figure 2-12 shows the corresponding COBOL data type and COBOL description for the arguments used in the numeric-formatting routines discussed in this section.

Figure 2-12
Correspondence Between Numeric-Formatting
Arguments and COBOL Data Types

Argument	COBOL Type	COBOL Description
FORMAT	CHARACTER	PIC X(LENGTH)
FORMAT-LEN	INTEGER	PIC S9(?) COMP
FVALUE	INTEGER	PIC S9(?) COMP
IVALUE	INTEGER	PIC S9(?) COMP
RESULT	CHARACTER	PIC X(LENGTH)
RESULT-LEN	INTEGER	PIC S9(?) COMP
STATUS	INTEGER	PIC S9(?) COMP

INFORMIX-ESQL/COBOL implements the PIC S9(?), shown in the COBOL Description column, as PIC S9(9) for a 4-byte integer or PIC S9(4) for a 2-byte integer.

Figure 2-13 shows the COMP equivalents for the types of COBOL compilers supported in ESQL/COBOL Version 7.2. Make sure you interpret the word COMP in the COBOL Description column as shown in Figure 2-13.

Figure 2-13
COMP Equivalents for COBOL Compilers

COMP Equivalent	Type of COBOL
COMP-2	MF COBOL/2
COMP-5	MF COBOL/2
COMP-1	RM/COBOL-85

Formatting Numeric Strings

The numeric expression format string consists of combinations of the * & # < , . - + () and \$ characters. Figure 2-14 describes those characters.

Figure 2-14
Description of Characters Used in Numeric Formatting

Character	Formatting Action
*	Fills with asterisks any positions in the display field that otherwise contains blanks.
&	Fills with zeros any positions in the display field that otherwise contains blanks.
#	Does not change any blank positions in the display field. Use this character to specify a maximum width for a field.
<	Left-justifies the numbers in the display field.
,	This character represents a literal. It displays as a comma but only when a number resides to its left.
.	This character represents a literal. It displays as a period. Only one period can exist in a format string.
-	This character represents a literal. It displays as a minus sign when <i>expr1</i> is less than zero. When you group several in a row, a single minus sign floats to the rightmost position without interfering with the number being printed.
+	This character represents a literal. It displays as a plus sign when <i>expr1</i> equals or exceeds zero and as a minus sign when less than zero. When you group several plus signs in a row, a single plus sign floats to the rightmost position without interfering with the number being printed.

(1 of 2)

Character	Formatting Action
(This character represents a literal. It displays as a left parenthesis before a negative number. It represents accounting parenthesis used in place of a minus sign to indicate a negative number. When you group several in a row, a single left parenthesis floats to the rightmost position without interfering with the number being printed.
)	This represents the accounting parenthesis used in place of a minus sign to indicate a negative number. A single one of these characters generally closes a format string that begins with a left parenthesis.
\$	This character represents a literal. It displays as a dollar sign. When you group several in a row, a single dollar sign floats to the rightmost position without interfering with the number being printed.

(2 of 2)

ESQL/COBOL allows the - + () and \$ characters to *float*. When a character floats, multiple leading occurrences of the character appear as a single character as far to the right as possible, without interfering with the displayed number.

When you use a nondefault locale, ECO-FFL uses the currency symbols that the locale defines. For more information, see Chapter 1 of the [Guide to GLS Functionality](#). ♦

Figure 2-15 shows example format strings for numeric expressions. The Formatted Result column uses the character b to represent a blank or space.

Figure 2-15
Format Strings, Values, and Results

Format String	Numeric Value	Formatted Result
"#####"	0	bbbbbb
"&&&&&"	0	00000
"\$\$\$\$\$"	0	bbbb\$
"*****"	0	*****
"<<<<<"	0	(null string)

(1 of 7)

GLS

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

(2 of 7)

Format String	Numeric Value	Formatted Result
"**,*!"	12345	12,345
"**,*!"	1234	*1,234
"**,*!"	123	***123
"**,*!"	12	****12
"**,*!"	1	*****1
"**,*!"	0	*****
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbbb.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbbb1.00
"&&,&&&.&&"	12345.67	12,345.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"\$\$,\$\$\$.\$\$"	12345.67	***** (overflow)
"\$\$,\$\$\$.\$\$"	1234.56	\$1,234.56
"\$\$,\$\$\$.##"	0.00	\$.00

Format String	Numeric Value	Formatted Result
"\$\$,\$\$\$,##"	1234.00	\$1,234.00
"\$\$,\$\$\$.&&"	0.00	\$.00
"\$\$,\$\$\$.&&"	1234.00	\$1,234.00
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"-#,###.##"	-12.34	-bbb12.34
"---,###.##"	-12.34	-bb12.34
"---,##.##"	-12.34	-12.34
"---,--.##"	-1.00	-1.00
"-##,###.##"	12345.67	12,345.67
"-##,###.##"	1234.56	1,234.56
"-##,###.##"	123.45	123.45
"-##,###.##"	12.34	12.34
"-#,###.##"	12.34	12.34
"---,###.##"	12.34	12.34
"---,##.##"	12.34	12.34
"---,---.##"	1.00	1.00
"---,---.---"	-.01	-.01
"---,---.&&"	-.01	-.01

(4 of 7)

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

(5 of 7)

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

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

(7 of 7)

ECO-FFL

Purpose

Use ECO-FFL to return a character-string representation of a 4-byte floating-point value for a given format.

Syntax

CALL ECO-FFL USING *FVALUE*, *FORMAT*, *FORMAT-LEN*, *RESULT*, *RESULT-LEN*, *STATUS*.

<i>FVALUE</i>	the 4-byte floating-point value that you provide
<i>FORMAT</i>	the character buffer of length <i>FORMAT-LEN</i> that you provide
<i>FORMAT-LEN</i>	the length that you specify for <i>FORMAT</i> (INTEGER)
<i>RESULT</i>	the character buffer of length <i>RESULT-LEN</i> that contains the result ECO-FFL returns
<i>RESULT-LEN</i>	the length that you specify for <i>RESULT</i> (INTEGER)
<i>STATUS</i>	the error status code (INTEGER) that ECO-FFL returns

Usage

Some of the codes (listed in the next section) returned in the *STATUS* parameter equal five characters in length. Thus, you can correctly identify these codes only when you define the *STATUS* variable as *S9(x)*, where *x* ≥ 5.

When you use a nondefault locale that has a multibyte code set, the ECO-FFL routine supports multibyte characters as formatting symbols. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

GLS

Return Codes

0	Success.
-1211	Insufficient memory.
-1217	The format string exceeds the specified size.

- | | |
|--------|---|
| -22234 | Insufficient buffer size. The ECO-FFL routine truncated the result to fit the buffer. |
| -22275 | INTERNAL ERROR: You exceeded the temporary buffer length. |

Example

The following code fragment shows how to call the ECO-FFL routine. The ECO-FFL routine works only with the Ryan-McFarland compiler. The Micro Focus compiler does not accept floating-point numbers for input into this routine.

```
.  
.   
.   
CALL ECO-FFL USING FVALUE, FORMAT, FORMAT-LEN,  
                  RESULT, RESULT-LEN, STAT-CODE.  
.   
.   
.
```

ECO-FIN

Purpose

Use ECO-FIN to return a character-string representation of an INTEGER value for a given format.

Syntax

CALL ECO-FIN USING IVALUE, FORMAT, FORMAT-LEN, RESULT, RESULT-LEN, STATUS.

IVALUE	the INTEGER value that you provide
FORMAT	the character buffer of length FORMAT-LEN that you provide
FORMAT-LEN	the length that you specify for FORMAT (INTEGER)
RESULT	the character buffer of length RESULT-LEN that contains the result ECO-FIN returns
RESULT-LEN	the length that you specify for RESULT (INTEGER)
STATUS	the error status code (INTEGER) that ECO-FIN returns

Usage

Some of the codes (listed in the next section) returned in the STATUS parameter equal five characters in length. Thus, you can correctly identify these codes only when you define the STATUS variable as S9(x), where x>=5.

When you use a nondefault locale that has a multibyte code set, the ECO-FIN routine supports multibyte characters as formatting symbols. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Return Codes

0	Success.
-1211	Insufficient memory.
-1217	The format string exceeds the specified size.

- 22234 Insufficient buffer size. The ECO-FIN routine truncated the result to fit the buffer.
- 22275 INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following code fragment from the ECOFIN program, accepts an integer value and formats it with a leading \$ sign, leading spaces, and decimal value.

```

1 *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOFIN.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 IVALUE PIC S9(9) COMP-5 VALUE 34567.
15 77 FORMAT PIC X(20) VALUE "$#####.###.&&".
16 77 FORMAT-LEN PIC S9(9) COMP-5 VALUE 20.
17 77 RESULT PIC X(20).
18 77 RESULT-LEN PIC S9(9) COMP-5 VALUE 20.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 EXEC SQL END DECLARE SECTION END-EXEC.
21
22
23          PROCEDURE DIVISION.
24          RESIDENT SECTION
25 *****
26
27          MAIN.
28 *****
29              DISPLAY 'THIS IS A TEST OF ECO-FIN.'.
30              CALL ECO-FIN USING IVALUE, FORMAT, FORMAT-LEN,
31                  RESULT, RESULT-LEN, STAT-CODE.
32              DISPLAY STAT-CODE.
33              DISPLAY IVALUE.
34              DISPLAY FORMAT.
35              DISPLAY RESULT.
36              STOP RUN.
```


Example Output

The output for the preceding code fragment displays the status plus the integer value designated for formatting, the format for the value, and the resulting formatted value with a dollar sign, leading spaces, and decimal.

```
THIS IS A TEST OF ECO-FIN.  
+0000000000  
+0000034567  
$#####.##.&&  
$          34,567.00
```


Working with Time Data Types

DATE Type Routines	3-3
ECO-DAT.	3-7
ECO-DAY.	3-10
ECO-DEF.	3-13
ECO-FMT.	3-18
ECO-JUL.	3-23
ECO-LYR.	3-26
ECO-MDY.	3-28
ECO-STR.	3-31
ECO-TDY.	3-33
DATETIME and INTERVAL Type Routines	3-35
ANSI SQL Standards for DATETIME and INTERVAL Values	3-36
ECO-DAI.	3-38
ECO-DSI.	3-42
ECO-DTC.	3-45
ECO-DTCVASC.	3-47
ECO-DTS.	3-52
ECO-DTTOASC.	3-57
ECO-DTX.	3-62
ECO-IDI.	3-65
ECO-IDN.	3-69
ECO-IMN.	3-73
ECO-INCVASC.	3-77
ECO-INTOASC.	3-82
ECO-INX.	3-87
ECO-IQU.	3-90
ECO-SQU.	3-93

T

his chapter describes Informix run-time routines used for time data type manipulation. The INFORMIX-ESQL/COBOL library extension includes those routines. When you use the **esqlcobol** compiler shell script, ESQL/COBOL automatically links the run-time routines.

Use those routines in your COBOL programs to convert and manipulate DATE, DATETIME, and INTERVAL data types. ESQL/COBOL divides those routines into nine DATE routines and fifteen DATETIME and/or INTERVAL routines.

This chapter discusses the purpose and use of Time data types in ESQL/COBOL. For more information on the DATE, DATETIME, and INTERVAL data types, refer to the [Informix Guide to SQL: Reference](#).

DATE Type Routines

This section describes the DATE type routines included in the libraries distributed with INFORMIX-ESQL/COBOL. Use these routines to convert dates written in string form to and from an internal format.

[Figure 3-1](#) shows the INFORMIX-ESQL/COBOL DATE routines and their descriptions. This section alphabetically lists and describes INFORMIX-ESQL/COBOL routines that manipulate DATE values.

Figure 3-1*ESQL/COBOL DATE Type Routines and their Descriptions*

Routine Name	What It Does
ECO-DAT	Converts an internal format to a string
ECO-DAY	Returns the day of the week
ECO-DEF	Converts a string to an internal format
ECO-FMT	Converts an internal format to a string
ECO-JUL	Returns the month, day, and year from an internal format
ECO-LYR	Determines whether a specified year equals a leap year
ECO-MDY	Returns an internal format from the month, day, and year
ECO-STR	Converts a string to an internal format
ECO-TDY	Returns the system date in an internal format

INFORMIX-ESQL/COBOL stores a date as a 4-byte INTEGER whose value equals the number of days occurring after December 31, 1899. Negative numbers represent dates before December 31, 1899, and positive numbers represent dates after December 31, 1899.

You can add and subtract numbers from DATE type values to produce a value corresponding to a date that many days later or earlier. You can also subtract two DATE types to get the number of days between them.

You must specify COBOL host variables that correspond to DATE data type columns as one of two PICTURE clauses:

- A DATE_TYPE, when you require a date in the form *mm/dd/yyyy*
- A 4-byte INTEGER, when you require a date in the form of the number of days occurring after December 31, 1899

When the following definitions appear in the DECLARE SECTION, the variable DATE-DISPLAY represents the date January 1, 1900 as the characters 01/01/1900, while the variable DATE-NUMERIC contains the integer value 1.

```
05 DATE-DISPLAY    DATE_TYPE.  
05 DATE-NUMERIC    PIC S9(8) COMP.
```

The following compiler-specific limitations exist for COBOL routines that manipulate DATE values:

- RM/COBOL-85 uses a 2-byte integer for a standard integer.
- MF COBOL/2 uses a 4-byte integer for a standard integer.

Figure 3-2 shows the correspondence between COBOL data types and arguments used in the DATE routines discussed in this section.

Figure 3-2
*Correspondences Between ESQL/COBOL
DATE Routines and COBOL Data Types*

Argument	COBOL Type
<i>DAY</i>	DAY OF WEEK
<i>FMT-LEN</i>	LENGTH
<i>JDATE</i>	DATE
<i>LEAP</i>	STATUS
<i>MDY</i>	MDY
<i>SDATE-LEN</i>	LENGTH
<i>STATUS</i>	STATUS
<i>TODAY</i>	DATE
<i>YEAR</i>	YEAR

Figure 3-3 shows the COBOL data types and their corresponding PICTURE declarations for the MF COBOL/2 and RM/COBOL-85 compilers.

Figure 3-3
*Correspondences Between COBOL Data Types and
COBOL Compiler PICTURE Declarations*

TYPE	MF COBOL/2 Compiler	RM/COBOL-85 Compiler
DAY OF WEEK	S9(9) COMP-5	S9(5) COMP-1
LENGTH	S9(9) COMP-5	S9(5) COMP-1
DATE	S9(9) COMP-5	S9(9) COMP-4
STATUS	S9(9) COMP-5	S9(5) COMP-1
YEAR	S9(9) COMP-5	S9(5) COMP-1
MDY	S9(9) COMP-5 OCCURS 3	S9(5) COMP-1 OCCURS 3

ECO-DAT

Purpose

Use ECO-DAT to convert a 4-byte INTEGER date to a character-string date of the form *mm/dd/yyyy*.

Syntax

```
CALL ECO-DAT USING JDATE, SDATE, SDATE-LEN.
```

<i>JDATE</i>	the 4-byte INTEGER input date that the ECO-DEF routine provides
<i>SDATE</i>	the character buffer of length <i>SDATE-LEN</i> that contains the result that ECO-DAT returns
<i>SDATE-LEN</i>	the length that you specify for <i>SDATE</i> (standard INTEGER)

Usage

ECO-DAT lacks a *STATUS* parameter. Instead, this routine sets the internal variable SQLCODE OF SQLCA. To check the SQLCA variable, call the ECO-GST or ECO-SQC routine. Refer to pages [2-20](#) and [2-21](#), respectively, for descriptions of those routines.

After you invoke the ECO-DAT routine, you can test SQLCODE with the ECO-GST or ECO-SQC routines to check SQLCODE for an error code. (Zero means no error). Refer to [Chapter 4, “Error Handling,”](#) for a discussion of error handling and the SQLCA record.

GLS

When you use a nondefault locale and do not set **DBDATE** or **GL_DATE**, the ECO-DAT routine uses the default date format that the locale defines. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

When you use a 2-digit year (yy) in a format, the ECO-DAT routine uses the setting of the **DBCENTURY** environment variable to determine the correct century to use. When you do not set **DBCENTURY**, ECO-DAT assumes the 20th century for 2-digit years. For information on how to set **DBCENTURY**, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

Example

The following code fragment from the ECODAT program, creates an INTEGER date with the ECO-DEF routine and converts that INTEGER to a character-string date with the ECO-DAT routine.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODAT.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 FMT PIC X(11) VALUE "mmm dd yyyy".
16 77 FMT-LEN PIC S9(9) COMP-5 VALUE 11.
17 77 SDATE PIC X(20) VALUE "Jan 15 1995".
18 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 11.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 *
21 77 SDATE-A PIC X(20).
22 77 SDATE-LEN-A PIC S9(9) COMP-5 VALUE 20.
23 EXEC SQL END DECLARE SECTION END-EXEC.
24
25
26 PROCEDURE DIVISION.
27 RESIDENT SECTION 1.
28 *****
29 MAIN.
30 *****
31 DISPLAY 'DATE STRING:          'SDATE.
32 DISPLAY 'CREATE INTEGER INPUT DATE.'.
33 CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE,
34     SDATE-LEN, STAT-CODE.
35 DISPLAY STAT-CODE.
36 DISPLAY JDATE
37 DISPLAY 'THIS IS A TEST OF ECODAT.'.
38 CALL ECO-DAT USING JDATE, SDATE-A, SDATE-LEN-A.
39 DISPLAY SDATE-A.
40 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status code, the output date (days occurring after December 31, 1899), and the character buffer for ECO-DEF in the format *mmm dd yyyy*, plus the input value (days occurring after December 31, 1899) and character buffer for ECO-DAT in the format *mm/dd/yyyy*.

```
DATE STRING: Jan 15 1995
CREATE INTEGER INPUT DATE.
+00000000000
+00000034713
THIS IS A TEST OF ECO-DAT.
01/15/1995
```

ECO-DAY

Purpose

Use ECO-DAY to return the day of the week represented as a standard INTEGER, given a 4-byte INTEGER date value.

Syntax

```
CALL ECO-DAY USING JDATE, DAY.
```

<i>JDATE</i>	the 4-byte INTEGER input date that the ECO-DEF routine provides
<i>DAY</i>	the value (standard INTEGER) that ECO-DAY returns

Return Values

0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Example

The following code fragment from the ECODAY program, produces an INTEGER date value using the ECO-DEF routine and sends that date to the ECO-DAY routine. Then, the ECO-DAY routine generates a numeric code for the day of the week.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODAY.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 FMT PIC X(11) VALUE "mmm dd yyyy".
16 77 FMT-LEN PIC S9(9) COMP-5 VALUE 11.
17 77 SDATE PIC X(20) VALUE "Jan 15 1995".
18 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 11.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 77 DAY CODE PIC S9(9) COMP-5.
21 EXEC SQL END DECLARE SECTION END-EXEC.
22
23
24 PROCEDURE DIVISION.
25 RESIDENT SECTION 1.
26 *****
27 MAIN.
28 *****
29 DISPLAY 'DATE STRING: ' SDATE.
30 DISPLAY 'PRODUCE AN INTEGER DATE VALUE.'.
31 CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE,
32     SDATE-LEN, STAT-CODE.
33 DISPLAY STAT-CODE.
34 DISPLAY JDATE.
35 DISPLAY 'THIS IS A TEST OF ECO-DAY.'.
36 CALL ECO-DAY USING JDATE, DAY-CODE.
37 DISPLAY 'DAY OF THE WEEK: ' DAY-CODE.
38 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status, the input date (days occurring after December 31, 1899), and day of the week (Monday for January 15th, 1995).

```
DATE STRING:  Jan 15 1995
PRODUCE AN INTEGER DATE VALUE.
+00000000000
+0000034713
THIS IS A TEST OF ECO-DAY.
DAY OF THE WEEK:  +00000000001
```

ECO-DEF

Purpose

Use ECO-DEF to create a 4-byte INTEGER date whose value represents the number of days occurring after December 31, 1899, for a character-string date format that you provide.

Syntax

```
CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE, SDATE-LEN, STATUS.
```

<i>JDATE</i>	the 4-byte INTEGER output date that ECO-DEF returns
<i>FMT</i>	the format character buffer of length <i>FMT-LEN</i> that you provide
<i>FMT-LEN</i>	the length that you specify for <i>FMT</i> (standard INTEGER)
<i>SDATE</i>	the character buffer of length <i>SDATE-LEN</i> that you provide
<i>SDATE-LEN</i>	the length that you specify for <i>SDATE</i> (standard INTEGER)
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-DEF returns

Usage

The string *FMT* uses the same formatting characters as ECO-FMT.

Make sure you specify the *JDATE* and the *FMT* string in the same sequential order relating to month, day, and year. They need not, however, use the same literals nor the same representation for month, day, and year.

[Figure 3-4](#) shows the valid combinations of *FMT* and *SDATE*.

Figure 3-4
Valid Values for FMT and SDATE Character Buffers

FMT	SDATE
"mmdyyy"	"Jan. 15th, 1995"
"mmm.dd.yyyy"	"Jan 15 1995"
"mmm.dd.yyyy"	"JAN-15-1995"
"mmdyy"	"011595"
"mmdyy"	"01/15/95"
"yy/mm/dd"	"95/01/15"
"yyyy/mm/dd"	"1995, January 15th"
"yyyy/mm/dd"	"In the year 1995, the month of January, its 15th day"
"dd-mm-yyyy"	"This 15th day of January, 1995"

Two of the codes, listed in the following section and returned in the *STATUS* parameter, equal six characters in length (including the minus sign). Thus, you can correctly identify these codes only when you define the *STATUS* variable as *S9(x)*, where *x* ≥ 6.

When you use a 2-digit year (yy) in a format, the ECO-DEF routine uses the setting of the **DBCENTURY** environment variable to determine the correct century to use. When you do not set **DBCENTURY**, ECO-DEF assumes the 20th century, as shown in the preceding table. For information on how to set **DBCENTURY**, see Chapter 4 of the *Informix Guide to SQL: Reference*.

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *FMT* argument of the ECO-DEF routine. For more information, see Chapter 6 of the *Guide to GLS Functionality*. ♦

GLS

Return Codes

0	Success.
-1204	You supplied an invalid year component in the <i>SDATE</i> parameter.
-1205	You supplied an invalid month component in the <i>SDATE</i> parameter.
-1206	You supplied an invalid day component in the <i>SDATE</i> parameter.
-1212	The <i>FMT</i> does not contain a month, day, and year component.
-22234	Insufficient buffer size. The ECO-DEF routine truncated the result to fit the buffer.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following code fragment from the ECODEF program, creates an INTEGER that represents the number of days occurring after December 31, 1899 for January 15th, 1995.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODEF.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 FMT PIC X(11) VALUE "mmm dd yyyy".
16 77 FMT-LEN PIC S9(9) COMP-5 VALUE 11.
17 77 SDATE PIC X(20) VALUE "Jan 15 1995".
18 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 11.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 EXEC SQL END DECLARE SECTION END-EXEC.
21
22
23 PROCEDURE DIVISION.
24 RESIDENT SECTION
25 *****
26 MAIN.
27 *****
28 DISPLAY 'DATE STRING: ' SDATE.
29 DISPLAY 'THIS IS A TEST OF ECO-DEF.'.
30 CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE,
31     SDATE-LEN, STAT-CODE.
32 DISPLAY STAT-CODE.
33 DISPLAY 'NUMBER OF DAYS SINCE 12/31/1899: ' JDATE.
34 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status and the output date (days occurring after December 31, 1899) for January 15th, 1995.

```
DATE STRING:  Jan 15 1995  
THIS IS A TEST OF ECO-DEF.  
+0000000000  
NUMBER OF DAYS SINCE 12/31/1899:  +0000034713
```

ECO-FMT

Purpose

Use ECO-FMT to convert a 4-byte INTEGER date to a character-string date formatted according to a pattern.

Syntax

CALL ECO-FMT USING <i>JDATE</i> , <i>FMT</i> , <i>FMT-LEN</i> , <i>SDATE</i> , <i>SDATE-LEN</i> , <i>STATUS</i> .	
<i>JDATE</i>	the 4-byte INTEGER input date that the ECO-DEF routine provides
<i>FMT</i>	the format character buffer of length <i>FMT-LEN</i> that you provide
<i>FMT-LEN</i>	the length that you specify for <i>FMT</i> (standard INTEGER)
<i>SDATE</i>	the character buffer of length <i>SDATE-LEN</i> that contains the formatted date that ECO-FMT returns
<i>SDATE-LEN</i>	the length that you specify for <i>SDATE</i> (standard INTEGER)
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-FMT returns

Usage

The string *FMT* consists of combinations of the characters *m*, *d*, and *y* as shown in the following table. The *SDATE* character buffer reproduces literally the characters that reside in *FMT*, but does not reproduce those characters shown in [Figure 3-5](#).

Figure 3-5
Characters That the SDATE Character Buffer Does Not Reproduce

Characters	Representation
dd	Day of the month as a 2-digit number (01-31)
ddd	Day of the week as a 3 letter abbreviation (Sun through Sat)
mm	Month as a 2-digit number (01-12)
mmm	Month as a 3-letter abbreviation (Jan through Dec)
yy	Year as a 2-digit number in the 1900s (00-99)
yyyy	Year as a 4-digit number (0001-9999)

Figure 3-6 converts the 4-byte INTEGER *JDATE* that corresponds to January 15, 1995, to a string *SDATE* using the format in *FMT*

Figure 3-6
Example of Date Conversion from 4-Byte Integer Format to SDATE Value Using Format Specified by FMT

FMT	SDATE
"mmddy"	"011595"
"ddmmy"	"150195"
"ymmdd"	"950115"
"yy/mm/dd"	"95/01/15"
"yy mm dd"	"95 01 15"
"yy-mm-dd"	"95-01-15"
"mmm. dd, yyyy"	"Jan. 15, 1995"
"mmm dd yyyy"	"Jan 15 1995"
"yyyy dd mm"	"1995 15 01"

FMT	SDATE
"mmm dd yyyy"	"Jan 15 1995"
"ddd, mmm. dd, yyyy"	"Sat, Jan. 15, 1995"
"(ddd) mmm. dd, yyyy"	"(Sat) Jan. 15, 1995"

(2 of 2)

Two of the codes, listed in the following section and returned in the *STATUS* parameter, equal six characters in length (including the minus sign). Thus, you can correctly identify these codes only when you define the *STATUS* variable as *S9(x)*, where *x* ≥ 6.

When you use a 2-digit year (*yy*) in a format, the ECO-FMT routine uses the setting of the **DBCENTURY** environment variable to determine the correct century to use. When you do not set **DBCENTURY**, ECO-FMT assumes the 20th century, as shown in the preceding table. For information on how to set **DBCENTURY**, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *FMT* argument of the ECO-FMT routine. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Return Codes

0	Success.
-1210	ECO-FMT cannot convert the internal date cannot to a month/day/year format.
-1211	Insufficient memory.
-22234	Insufficient buffer size. The ECO-FMT routine truncated the result to fit the buffer.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following code fragment from the ECOFMT program, creates an INTEGER date with the ECO-DEF routine and converts that integer to a character string formatted to a specific pattern using the ECO-FMT routine.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4  ECOFMT.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 FMT PIC X(11) VALUE "mmm dd yyyy".
16 77 FMT-LEN PIC S9(9) COMP-5 VALUE 11.
17 77 SDATE PIC X(20) VALUE "Jan 15 1995".
18 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 11.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 *
21 77 FMT-A PIC X(8) VALUE "yy mm dd".
22 77 FMT-LEN-A PIC S9(9) COMP-5 VALUE 8.
23 77 SDATE-A PIC X(20).
24 77 SDATE-LEN-A PIC S9(9) COMP-5 VALUE 20.
25 77 STAT-CODE-A PIC S9(9) COMP-5.
26 EXEC SQL END DECLARE SECTION END-EXEC.
27
28 PROCEDURE DIVISION.
29 RESIDENT SECTION 1.
30 *****
31 MAIN.
32 *****
33 DISPLAY 'DATE VALUE: ' SDATE.
34 DISPLAY 'CREATE AN INTEGER DATE.'.
35 CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE,
36 SDATE LEN, STAT-CODE.
37 DISPLAY JDATE.
38 DISPLAY 'THIS IS A TEST OF ECO-FMT.'.
39 CALL ECO-FMT USING JDATE, FMT-A, FMT-LEN-A, SDATE-A,
40 SDATE-LEN-A, STAT-CODE-A.
41 DISPLAY STAT-CODE-A.
42 DISPLAY 'FORMAT: ' FMT-A.

```

```
43  DISPLAY SDATE-A.  
44  STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status code, the output value (days occurring after December 31, 1899), and the character buffer for ECO-DEF in the format *mmm dd yyyy*, plus the status code, the input value (days occurring after December 31, 1899), and the character buffer for ECO-FMT in the format *yy mm dd*.

```
DATE VALUE:  Jan 15 1995  
CREATE AN INTEGER DATE  
+0000034713  
THIS IS A TEST OF ECO-FMT.  
+0000000000  
FORMAT:  yy mm dd  
95 01 15
```

ECO-JUL

Purpose

Use ECO-JUL to create an array of three standard INTEGER values that contain the month, day, and year components corresponding to a 4-byte INTEGER date.

Syntax

```
CALL ECO-JUL USING JDATE, MDY, STATUS.
```

<i>JDATE</i>	the 4-byte INTEGER input date that the ECO-DEF routine provides
<i>MDY</i>	the array containing month, day, year (standard INTEGER) that ECO-JUL returns
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-JUL returns

Return Codes

= 0	Success.
< 0	Failure.

Example

This following code fragment from the ECOJUL program, breaks out the month, day, and year components for the input date.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4  ECOJUL.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 FMT PIC X(11) VALUE "mmm dd yyyy".
16 77 FMT-LEN PIC S9(9) COMP-5 VALUE 11.
17 77 SDATE PIC X(20) VALUE "Jan 15 1995".
18 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 11.
19 77 STAT-CODE PIC S9(9) COMP-5.
20 *
21 01 MON-DAY-YR.
22 02 MDY PIC S9(9) COMP-5 OCCURS 3 TIMES.
23 77 STAT-CODE-A PIC S9(9) COMP-5.
24 EXEC SQL END DECLARE SECTION END-EXEC.
25
26
27 PROCEDURE DIVISION.
28 RESIDENT SECTION 1.
29 *****
30 MAIN.
31 *****
32 DISPLAY 'DATE VALUE: ' SDATE.
33 DISPLAY 'CREATE AN INTEGER DATE.'.
34 CALL ECO-DEF USING JDATE, FMT, FMT-LEN, SDATE,
35 SDATE-LEN, STAT-CODE.
36 DISPLAY JDATE.
37 DISPLAY 'THIS IS A TEST OF ECO-JUL.'.
38 CALL ECO-JUL USING JDATE, MON-DAY-YR, STAT-CODE-A.
39 DISPLAY STAT-CODE-A.
40 DISPLAY 'MONTH: ' MDY(1).
41 DISPLAY 'DAY: ' MDY(2).
42 DISPLAY 'YEAR: ' MDY(3).
43 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status code and output date (days occurring after December 31, 1899) for ECO-DEF, and the status code, input date (days occurring after December 31, 1899), and month, day, and year components for the ECO-JUL date.

```
DATE VALUE:  Jan 15 1995
CREATE AN INTEGER DATE.
+0000034713
THIS IS A TEST OF ECO-JUL.
+0000000000
MONTH:  +0000000001
DAY:    +0000000015
YEAR    +0000001995
```

ECO-LYR

Purpose

Use ECO-LYR to return 1 (TRUE) when *YEAR* equals a leap year and 0 (FALSE) when *YEAR* does not equal a leap year.

Syntax

```
CALL ECO-LYR USING YEAR, LEAP.
```

<i>YEAR</i>	the year (standard INTEGER) that you provide
<i>LEAP</i>	the leap year indicator (standard INTEGER) that ECO-LYR returns

Usage

The argument *YEAR* must contain only the year component of a date and not the date itself.

You must express *YEAR* using a full year (such as 1996) and not an abbreviated year (such as 96).

Return Codes

The ECO-LYR routine returns one of the following codes in *LEAP*:

1 (TRUE)	represents a leap year.
0 (FALSE)	represents a non-leap year.

Example

The following code fragment from the ECOLYR program, takes an INTEGER year that you provide and returns an INTEGER value representing a leap year or no leap year.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOLYR.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 YEAR-NUM PIC S9(9) COMP-5 VALUE 1996.
15 77 LEAP-INDICATOR PIC S9(9) COMP-5.
16 EXEC SQL END DECLARE SECTION END-EXEC.
17
18
19 PROCEDURE DIVISION.
20 RESIDENT SECTION 1.
21 *****
22 MAIN.
23 *****
24 DISPLAY 'THIS IS A TEST OF ECO-LYR.'.
25 CALL ECO-LYR USING YEAR-NUM, LEAP-INDICATOR.
26 DISPLAY 'YEAR VALUE: 'YEAR-NUM.
27 DISPLAY 'LEAP YEAR : 'LEAP-INDICATOR.
28 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the year 1996 and returns the *LEAP* code 1 to indicate that 1996 was leap year.

```

THIS IS A TEST OF ECO-LYR.
YEAR VALUE:  +0000001996
LEAP YEAR :  +0000000001
```

ECO-MDY

Purpose

Use ECO-MDY to create a 4-byte INTEGER date from an array of three standard INTEGER values that contain numeric values for the month, day, and year.

Syntax

```
CALL ECO-MDY USING MDY, JDATE, STATUS.
```

<i>MDY</i>	the array containing month, day, year (standard INTEGER) that you provide
<i>JDATE</i>	the 4-byte INTEGER output date that ECO-MDY returns
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-MDY returns

Usage

You must specify the year as a full year (such as 1994) and not as an abbreviated year (such as 94).

Two of the codes, listed in the following section and returned in the *STATUS* parameter, equal six characters in length (including the minus sign). Thus, you can correctly identify these codes only when you define the *STATUS* variable as *S9(x)*, where $x \geq 6$.

Return Codes

0	Success.
-1204	An invalid year component resides in <i>MDY</i> (2).
-1205	An invalid month component resides in <i>MDY</i> (0).
-1206	An invalid day component resides in <i>MDY</i> (1).
-22234	Insufficient buffer size. The ECO-MDY routine truncated the result to fit the buffer.

-22275

INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following code fragment from the ECOMDY program, takes an array containing month, year, and day INTEGER values and creates an INTEGER date using the ECO-MDY routine.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4  ECOMDY.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 01 MON-DAY-YR.
16 02 MDY PIC S9(9) COMP-5 OCCURS 3 TIMES.
17 77 STAT-CODE PIC S9(9) COMP-5.
18 EXEC SQL END DECLARE SECTION END-EXEC.
19
20
21 PROCEDURE DIVISION.
22 RESIDENT SECTION 1.
23 *****
24 MAIN.
25 *****
26 DISPLAY 'THIS IS A TEST OF ECO-MDY.'.
27 MOVE 01 TO MDY(1).
28 MOVE 15 TO MDY(2).
29 MOVE 1995 TO MDY(3).
30 DISPLAY 'MONTH VALUE:  ' MDY(1).
31 DISPLAY 'DAY VALUE:    ' MDY(2).
32 DISPLAY 'YEAR VALUE:   ' MDY(3).
33 CALL ECO-MDY USING MON-DAY-YR, JDATE, STAT-CODE.
34 DISPLAY STAT-CODE.
35 DISPLAY 'INTEGER REPRESENTATION:  'JDATE.
36 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the status code and INTEGER input date (days occurring after December 31, 1899) created from a month, day, and year array.

```
THIS IS A TEST OF ECO-MDY.  
MONTH VALUE:  +0000000001  
DAY VALUE:    +0000000015  
YEAR VALUE    +0000001995  
+0000000000  
INTEGER REPRESENTATION:  0000034713
```


ECO-STR

Purpose

Use ECO-STR to convert a character-string date to a 4-byte INTEGER date.

Syntax

```
CALL ECO-STR USING SDATE, SDATE-LEN, JDATE, STATUS.
```

<i>SDATE</i>	the character buffer of length <i>SDATE-LEN</i> that you provide
<i>SDATE-LEN</i>	the length that you specify for <i>SDATE</i> (standard INTEGER)
<i>JDATE</i>	the 4-byte INTEGER output date that ECO-STR returns
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-STR returns

Usage

SDATE must contain a numeric month, day, and year, in that order. Use a hyphen, backslash, or period as a separator between the month, day, and year. You can specify the year as two or four characters (for example, 94 or 1994).

GLS

When you use a nondefault locale and do not set **DBDATE** or **GL_DATE**, the ECO-STR routine uses the default date format that the locale defines. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

When you use a 2-digit year (yy) in a format, the ECO-STR routine uses the setting of the **DBCENTURY** environment variable to determine the correct century to use. When you do not set **DBCENTURY**, ECO-STR assumes the 20th century for 2-digit years. For information on how to set **DBCENTURY**, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

Return Codes

= 0	Success.
< 0	Failure.

Example

The following code fragment from the ECOSTR program converts a character string date into an INTEGER date using the routine ECO-STR.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOSTR.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 JDATE PIC S9(9) COMP-5.
15 77 SDATE PIC X(10) VALUE "01-15-1995".
16 77 SDATE-LEN PIC S9(9) COMP-5 VALUE 10.
17 77 STAT-CODE PIC S9(9) COMP-5.
18 EXEC SQL END DECLARE SECTION END-EXEC.
19
20
21 PROCEDURE DIVISION.
22 RESIDENT SECTION 1.
23 *****
24 MAIN.
25 *****
26 DISPLAY 'THIS IS A TEST OF ECO-STR.'.
27 DISPLAY 'DATE STRING VALUE: ' SDATE.
28 CALL ECO-STR USING SDATE, SDATE-LEN, JDATE, STAT-CODE.
29 DISPLAY STAT-CODE.
30 DISPLAY 'INTEGER REPRESENTATION: ' JDATE.
31 STOP RUN.

```

Example Output

The output from the preceding code fragment displays the status and output date (days occurring after December 31, 1899).

```

THIS IS A TEST OF ECO-STR.
DATE STRING VALUE:  01-15-1995
+0000000000
INTEGER REPRESENTATION:  +0000034713

```

ECO-TDY

Purpose

Use ECO-TDY to put the system date into a 4-byte INTEGER date.

Syntax

```
CALL ECO-TDY USING TODAY.
```

TODAY the 4-byte INTEGER output date

Usage

You move the system date into *TODAY*. When you call the ECO-TDY routine, ECO-TDY converts the data contained in *TODAY* into a 4-byte INTEGER date.

Example

The following code fragment accepts the system date and displays that date as an integer date using the ECO-TDY routine.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOTDY.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION ECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 TODAY PIC S9(9) COMP-5.
15 EXEC SQL END DECLARE SECTION END-EXEC.
16
17
18 PROCEDURE DIVISION.
19 RESIDENT SECTION 1.
20 *****
21 MAIN.
22 *****
23 DISPLAY 'THIS IS A TEST OF ECO-TDY.'.
24 ACCEPT TODAY FROM DATE.
25 DISPLAY 'TODAY  ' TODAY.
26 CALL ECO-TDY USING TODAY.
27 DISPLAY 'INTEGER REPRESENTATION:  'TODAY.
28 STOP RUN.
```

Example Output

The preceding code fragment displays the integer output date (days occurring after December 31, 1899). The following output assumes that January 15th, 1995 is the current system date.

```

THIS IS A TEST OF ECO-TDY.
TODAY:  +0000950115
INTEGER REPRESENTATION:  +0000034713
```

DATETIME and INTERVAL Type Routines

The DATETIME data type stores an instant in time expressed as a calendar date and time of day. The INTERVAL data type stores a value that represents a span of time. That time span can encompass either a span of years and months or a span of days, hours, minutes, seconds, and fractions of a second.

INFORMIX-ESQL/COBOL provides a number of built-in routines to manipulate DATETIME and INTERVAL values. DATETIME and INTERVAL exist as character buffers that contain a DATETIME or INTERVAL ASCII string literal.

Figure 3-7 lists the routines included in the ESQL/COBOL package that manipulate character strings in proper DATETIME and INTERVAL format. This section alphabetically lists and discusses those routines.

Figure 3-7
Descriptions of ESQL/COBOL DATETIME and INTERVAL Routines

Routine Name	What It Does
ECO-DAI	Adds an INTERVAL string to a DATETIME string
ECO-DSI	Subtracts an INTERVAL value from a DATETIME value
ECO-DTC	Determines the current DATETIME value
ECO-DTCVASC	Converts specified format string to ANSI DATETIME format
ECO-DTS	Subtracts two DATETIME strings
ECO-DTTOASC	Converts ANSI DATETIME string to specified format
ECO-DTX	Extends a DATETIME value to a different qualifier
ECO-IDI	Divides an INTERVAL value by an INTERVAL value
ECO-IDN	Divides an INTERVAL value by a numeric value
ECO-IMN	Multiplies an INTERVAL value with a numeric value
ECO-INCVASC	Converts specified format string to ANSI INTERVAL format
ECO-INTOASC	Converts ANSI INTERVAL string to specified ASCII format

(1 of 2)

Routine Name	What It Does
ECO-INX	Extends an INTERVAL value to a different qualifier
ECO-IQU	Determines INTEGER qualifier for character-string qualifier
ECO-SQU	Determines character-string qualifier for INTEGER qualifier

(2 of 2)

The following sections discuss the DATETIME and INTERVAL data types described in Figure 3-7. The preceding routines use an implementation-defined INTEGER. The INTEGER length equals 4-bytes when supported; otherwise, the INTEGER length equals 2-bytes.



Important: You must set the **DBTIME** environment variable before calling *ECO-DTCVASC* or *ECO-DTTOASC* for these routines to work properly. These routines support DATETIME information for the parts of the world that do not follow the ANSI SQL standard for representing date and time. When you do not set **DBTIME**, your program uses the default **DBTIME** setting. Refer to the “[Informix Guide to SQL: Reference](#)” for information on how to set **DBTIME**.

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 shown in the following example:

YYYY-MM-DD HH:MM:SS

The standards for an INTERVAL value specify two different classes of intervals known as *year to month* and *day to fraction*. The following example shows the format of the *year to month* class:

YYYY-MM

You can also consider a subset of the preceding format as valid (for example, just a month interval).

The following example shows the format of the *day to fraction* class:

```
DD HH:MM:SS.F
```

You can consider any subset of contiguous fields as valid (for example, *minute to fraction*).

You can also set the **DBTIME** environment variable to specify a format that differs from the ANSI standards (For example, a format that uses different delimiters). For information on **DBTIME**, refer to the [Informix Guide to SQL: Reference](#).

ECO-DAI

Purpose

Use ECO-DAI to add an INTERVAL value to a DATETIME value. ECO-DAI stores the resulting DATETIME value in *RESULT*.

Syntax

```
CALL ECO-DAI USING DATE, DATE-LEN, DATE-QUAL, INV, INV-LEN, INV-QUAL, RESULT, RESULT-LEN, STATUS.
```

<i>DATE</i>	the DATETIME value that you provide
<i>DATE-LEN</i>	the length that you specify for <i>DATE</i> (INTEGER)
<i>DATE-QUAL</i>	the qualifier of <i>DATE</i> (INTEGER) that the ECO-IQU routine provides
<i>INV</i>	the INTERVAL value that you provide
<i>INV-LEN</i>	the length that you specify for <i>INV</i> (INTEGER)
<i>INV-QUAL</i>	the qualifier of <i>INV</i> (INTEGER) that the ECO-IQU routine provides
<i>RESULT</i>	the resultant DATETIME value that ECO-DAI returns
<i>RESULT-LEN</i>	the length that you specify for <i>RESULT</i> (INTEGER)
<i>STATUS</i>	the error status code (INTEGER) that ECO-DAI returns

Usage

Two of the codes, listed in the following section and returned in the *STATUS* parameter, equal six characters in length (including the minus sign). Thus, you can correctly identify these codes only when you define the *STATUS* variable as *S9(x)*, where $x \geq 6$.

Return Codes

0	Success.
-1204	An invalid DATETIME value resides in the <i>DATE</i> parameter.
-1266	You cannot use incompatible INTERVAL and/or DATETIME values.
-1267	The result of the DATETIME computation exceeds the allowed range.
-22234	Insufficient buffer size. The ECO-DAI routine truncated the result to fit the buffer.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following code fragment from the ECODAI program tests the ECO-DAI routine. The code fragment accepts a DATETIME and INTERVAL qualifier from the routine ECO-IQU, adds an INTERVAL value to a DATETIME value, and displays the result.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4  ECODAI.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 QTYPE PIC S9(9) COMP-5 VALUE 0.
15 77 SQUAL PIC X(30) VALUE "year to month".
16 77 SQUAL-LEN PIC S9(9) COMP-5 VALUE 30.
17 77 IQUAL PIC S9(9) COMP-5.
18 77 STAT-CODE-A PIC S9(9) COMP-5.
19 *
20 77 ZQUAL PIC S9(9) COMP-5.
21 *
22 77 DATE-VAL PIC X(30) VALUE "1995-7".
23 77 DATE-LEN PIC S9(9) COMP-5 VALUE 30.
24 77 DATE-QUAL PIC S9(9) COMP-5.
25 77 INV PIC X(30) VALUE "1-3".
26 77 INV-LEN PIC S9(9) COMP-5 VALUE 30.
27 77 INV-QUAL PIC S9(9) COMP-5.
28 77 RESULT PIC X(30).
29 77 RESULT-LEN PIC S9(9) COMP-5 VALUE 30.
30 77 STAT-CODE-B PIC S9(9) COMP-5.
31 EXEC SQL END DECLARE SECTION END-EXEC.
32
33 PROCEDURE DIVISION.
34 RESIDENT SECTION 1.
35 *****
36 MAIN.
37 *****
38 DISPLAY 'INITIALIZE BOTH VALUES OF ECO-IQU.'.
39 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, IQUAL,
40     STAT-CODE-A.
41 DISPLAY IQUAL.

```

```

42  MOVE 1 TO QTYPE.
43  MOVE "year to month" TO SQUAL.
44  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, ZQUAL,
45      STAT-CODE-A.
46  DISPLAY ZQUAL.
47  MOVE IQUAL TO DATE-QUAL.
48  MOVE ZQUAL TO INV-QUAL.
49  DISPLAY 'THIS IS A TEST OF ECO-DAI.'.
50  CALL ECO-DAI USING DATE-VAL, DATE-LEN, DATE-QUAL,
51      INV, INV-LEN, INV-QUAL, RESULT,
52      RESULT-LEN, STAT-CODE-B.
53  DISPLAY STAT-CODE-B.
54  DISPLAY 'DATETIME VALUE:  ' DATE-VAL.
55  DISPLAY 'INTERVAL VALUE:  ' INV.
56  DISPLAY 'RESULT OF ADDITION:  ' RESULT.
57  STOP RUN.

```

Example Output

The output for the preceding code fragment displays the initialized values for the INTERVAL and DATETIME qualifiers from ECO-IQU, plus the status code, *year-to-month* DATETIME value, DATETIME qualifier, INTERVAL value, INTERVAL qualifier, and the result of the addition of an INTERVAL value to a DATETIME value for ECO-DAI.

```

INITIALIZE BOTH VALUES OF ECO-IQU.
+0000001538
+0000001538
THIS IS A TEST OF ECO-DAI.
+0000000000
DATETIME VALUE:  1995-7
INTERVAL VALUE:  1-3
RESULT OF ADDITION:  1996-10

```

ECO-DSI

Purpose

Use ECO-DSI to subtract an INTERVAL value from a DATETIME value.

Syntax

```
CALL ECO-DSI USING DT, DT-LEN, DT-QUAL, INV, INV-LEN, INV-QUAL,
DTRES, DTRES-LEN, STATUS.
```

<i>DT</i>	the DATETIME value that you provide
<i>DT-LEN</i>	the length that you specify for <i>DT</i> (standard INTEGER)
<i>DT-QUAL</i>	the qualifier of <i>DT</i> (standard INTEGER) that the ECO-IQU routine provides
<i>INV</i>	the INTERVAL value that you provide
<i>INV-LEN</i>	the length that you specify for <i>INV</i> (standard INTEGER)
<i>INV-QUAL</i>	the qualifier of <i>INV</i> (standard INTEGER) that the ECO-IQU routine provides
<i>DTRES</i>	the resultant DATETIME value that ECO-DSI returns
<i>DTRES-LEN</i>	the length that you specify for <i>DTRES</i> (standard INTEGER)
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-DSI returns

Usage

The ECO-DSI routine subtracts the INTERVAL value in *INV* from the DATETIME value in *DT*. The ECO-DSI routine stores the resulting DATETIME value in *DTRES*.

DTRES inherits the *DT* qualifier.

You must set the INTERVAL value to either the *year-to-month* or *day-to-fraction(5)* range.

Make sure you include all the fields present in the INTERVAL value in the DATETIME value.

Return Codes

= 0 Success.
 < 0 Failure.

Example

The following code fragment from the ECODSI program accepts a DATETIME and an INTERVAL qualifier from ECO-IQU, subtracts an INTERVAL value from a DATETIME value, and displays the result.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODSI.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE PIC S9(9) COMP-5 VALUE 0.
18 77 SQUAL PIC X(30) VALUE "year to month".
19 77 SQUAL-LEN PIC S9(9) COMP-5 VALUE 30.
20 77 STAT-CODE-A PIC S9(9) COMP-5.
21 77 DATE-VAL PIC X(30) VALUE "1995-10".
22 77 DATE-LEN PIC S9(9) COMP-5 VALUE 30.
23 77 DATE-QUAL PIC S9(9) COMP-5.
24 77 INV PIC X(30) VALUE "1-3".
25 77 INV-LEN PIC S9(9) COMP-5 VALUE 30.
26 77 INV-QUAL PIC S9(9) COMP-5.
27 77 DTRES PIC X(30).
28 77 DTRES-LEN PIC S9(9) COMP-5 VALUE 30.
29 77 STAT-CODE-B PIC S9(9) COMP-5.
30 EXEC SQL END DECLARE SECTION END-EXEC.
31 *
32 PROCEDURE DIVISION.
33 RESIDENT SECTION 1.
34 *
```

```

35  *
36  *Begin Main routine. Initialize both values of
37  *ECO-IQU producing two integer qualifiers; an
38  *INTERVAL and a DATETIME value.
39  *Use the values generated by ECO-IQU
40  *as input values for ECO-DSI. Display the
41  *resultant DATETIME value.
42  *
43  MAIN.
44  DISPLAY 'INITIALIZE BOTH VALUES OF ECO-IQU.'.
45  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN,
46  DATE QUAL, STAT-CODE-A.
47  DISPLAY DATE-QUAL.
48  MOVE 1 TO QTYPE.
49  MOVE "year to month" to SQUAL.
50  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, INV-QUAL,
51  STAT-CODE-A.
52  DISPLAY INV-QUAL.
53  *
54  DISPLAY 'THIS IS A TEST OF ECO-DSI.'
55  CALL ECO-DSI USING DATE-VAL, DATE-LEN, DATE-QUAL, INV,
56  INV-LEN, INV-QUAL, DTRES, DTRES-LEN, STAT-CODE-B.
57  DISPLAY STAT-CODE-B.
58  DISPLAY 'DATETIME VALUE: ' DATE-VAL.
59  DISPLAY 'INTERVAL VALUE: ' INV.
60  DISPLAY 'RESULT OF SUBTRACTION: ' DTRES.
61  DISPLAY ' '.
62  STOP RUN.
63  *

```

Example Output

The output for the preceding code fragment displays the status, the input date, the DATETIME qualifier, the INTERVAL value, the INTERVAL qualifier, and the output date (the input date minus the INTERVAL value).

```

INITIALIZE BOTH VALUES OF ECO-IQU.
+0000001538
+0000001538
THIS IS A TEST OF ECO-DSI.
+0000000000
DATETIME VALUE: 1995-10
INTERVAL VALUE: 1-3
RESULT OF SUBTRACTION: 1994-07

```

ECO-DTC

Purpose

Use ECO-DTC to determine the current DATETIME value.

Syntax

```
CALL ECO-DTC USING DATE, DATE-LEN, STATUS.
```

<i>DATE</i>	the DATETIME value. You place the system date into <i>DATE</i> with an ACCEPT statement. ECO-DTC changes the contents of <i>DATE</i> into a DATETIME value.
<i>DATE-LEN</i>	the length that you specify for <i>DATE</i> (INTEGER)
<i>STATUS</i>	the error status code (INTEGER) that ECO-DTC returns

Return Codes

= 0	Success.
< 0	Failure.

Example

The following code fragment from the ECODTC program accepts the current date and determines the current DATETIME value.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODTC.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 DATE-VAL PIC X(30).
15 77 DATE-LEN PIC S9(9) COMP-5 VALUE 30.
16 77 STAT-CODE PIC S9(9) COMP-5.
17 EXEC SQL END DECLARE SECTION END-EXEC.
18
19
20 PROCEDURE DIVISION.
21 RESIDENT SECTION 1.
22 *****
23 MAIN.
24 *****
25 DISPLAY 'THIS IS A TEST OF ECO-DTC.'.
26 ACCEPT DATE-VAL FROM DATE.
27 CALL ECO-DTC USING DATE-VAL, DATE-LEN, STAT-CODE.
28 DISPLAY 'CURRENT DATE:  ' DATE-VAL.
29 DISPLAY STAT-CODE.
30 STOP RUN.
```

Example Output

The output for the preceding code fragment displays the DATETIME value for the input date, January 15th, 1995, and the status.

```

THIS IS A TEST OF ECO-DTC.
CURRENT DATE:  1995-01-15 08:53:00.630
+0000000000
```


ECO-DTCVASC

Purpose

Use ECO-DTCVASC to convert a string with a specified format to an ANSI DATETIME string.

Syntax

```
CALL ECO-DTCVASC USING STR, STR-LEN, FMTSTR, FMTSTR-LEN, DT, DT-LEN, DT-QUAL, STATUS.
```

<i>STR</i>	the address of the input DATETIME string in the format of <i>FMTSTR</i> that you provide
<i>STR-LEN</i>	the length that you specify for <i>STR</i>
<i>FMTSTR</i>	the address of the format string for the input (<i>STR</i>), using the directives defined for DBTIME . When you leave the <i>FMTSTR</i> string empty, the ECO-DTCVASC routine uses the format that the DBTIME environment variable specifies. When you do not set DBTIME and leave the <i>FMTSTR</i> string empty, the ECO-DTCVASC routine uses the ANSI SQL format, causing unpredictable results. For information on how to set DBTIME , refer to the Informix Guide to SQL: Reference . You provide the value for <i>FMTSTR</i> .
<i>FMTSTR-LEN</i>	the length that you specify for <i>FMTSTR</i>
<i>DT</i>	the address of the resulting ANSI DATETIME string that ECO-DTVASC returns
<i>DT-LEN</i>	the length that you specify for <i>DT</i>
<i>DT-QUAL</i>	the qualifier for <i>DT</i> that the ECO-IQU routine provides
<i>STATUS</i>	the error status code that ECO-DTCVASC returns

Usage

The input string can contain leading and trailing spaces. However, from the first to the last significant digit, ECO-DTCVASC accepts only digit and delimiter characters appropriate to the fields that the format string implies.

When calling ECO-DTCVASC, make sure you use only contiguous fields in the DATETIME input string. In other words, when you specify an *hour-to-second* qualifier, make sure the values for hour, minute, and second reside in the string (not necessarily in that order) or an error results.

The ECO-DTCVASC routine does not require you to make the output qualifier match the input qualifier that the format string specifies. When you use an output qualifier that differs from the input qualifier, ECO-DTCVASC performs extensions as shown in the following list:

- Discards *STR* fields not included in *DT*
- Uses the current time and date to fill in fields to the left of the most-significant field in *STR*
- Uses zeros to fill in fields to the right of the least-significant field in *STR*

If you specify a valid input string and format specification, ECO-DTCVASC sets the output value returns zero in *STATUS*. Otherwise, ECO-DTCVASC returns an error code and the output string produces unpredictable results.

If *DT* lacks sufficient size, ECO-DTCVASC truncates the result and returns an error.

One of the codes, listed in the following section and returned in the *STATUS* parameter, equals six characters in length (including the minus sign). Thus, you can correctly identify this code only when you define the *STATUS* variable as *S9(x)*, where $x \geq 6$.

Return Codes

-1211	Insufficient memory.
-1260	You cannot convert between the specified types.
-1261	The first field of a DATETIME or INTERVAL value contains too many digits.
-1262	A nonnumeric character resides in a DATETIME or INTERVAL value.
-1263	An incorrect or out of range field resides in a DATETIME or INTERVAL value.
-1264	Extra characters exist at the end of a DATETIME or INTERVAL value.
-1265	An overflow occurred on a DATETIME or INTERVAL operation.
-1266	Incompatible INTERVAL or DATETIME values exist.
-1267	A DATETIME computation result exceeds the allowed range.
-1268	Invalid DATETIME qualifier.
-1271	Missing decimal point in fraction.
-1272	You did not specify an input buffer.
-1273	The output buffer either cannot hold the result due to insufficient size or contains a null value.
-1275	Invalid field width for a DATETIME or INTERVAL format string.
-1276	Unsupported format conversion character.
-1277	Input does not match format specification.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following two example code fragments convert a string with a specified format to an ANSI DATETIME string. Both code fragments initialize the *DT* argument to the date of a fictional birthday party. The input and output qualifiers are the same (month to minute).

```

1 *The input and output qualifiers are the same (month to minute)
2
3 MOVE "June 9 at 01:30pm"          TO STR.
4 MOVE 17                          TO STR-LEN.
5
6 *Note the absence of field-width and precision specification
7 *in the input format string.
8 MOVE "%B %d at %I:%M%p"          TO FMTSTR.
9 MOVE 16                          TO FMTSTR-LEN.
10
11 MOVE 0                          TO FLAG.
12 MOVE "MONTH TO MINUTE"          TO QUAL.
13 MOVE 15                          TO QUAL-LEN.
14
15 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, DT-QUAL, STAT.
16 IF STAT < 0
17   DISPLAY 'DT-QUAL ERROR ', STAT.
18
19 MOVE 50                          TO DT-LEN.
20
21 *'DT' will be set to "06-09 13:30"
22
23 CALL ECO-DTCVASC USING STR, STR-LEN, FMTSTR, FMTSTR-LEN,
24   DT, DT-LEN, DT-QUAL, STAT.
25 IF STAT < 0
26   DISPLAY 'DT-CVASC ERROR ', STAT.
27 *The input and output qualifiers are different:
28 *input qual : month to minute
29 *output qual: year to minute
30
31 MOVE "June 9 at 01:30pm"          TO STR.
32 MOVE 17                          TO STR-LEN.
33
34 MOVE "%B %d at %I:%M%p"          TO FMTSTR.
35 MOVE 16                          TO FMTSTR-LEN.
36 MOVE 0                          TO FLAG.
37 MOVE "YEAR TO MINUTE"           TO QUAL.
38 MOVE 14                          TO QUAL-LEN.
39
40 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, DT-QUAL, STAT.
41 IF STAT < 0
42   DISPLAY 'DT-QUAL ERROR ', STAT.
43
44 MOVE 50                          TO DT-LEN.
45 *'DT' will be set to "XXXX-06-09 13:30"
46 *Notice that the output has been extended, and the year

```

```
47 *is set to current year.  
48  
49 CALL ECO-DTCVASC USING STR, STR-LEN, FMTSTR, FMTSTR-LEN,  
50     DT, DT-LEN, DT-QUAL, STAT.  
51 IF STAT < 0  
52     DISPLAY 'DT-CVASC ERROR ', STAT.
```

ECO-DTS

Purpose

Use ECO-DTS to subtract two DATETIME values. The ECO-DTS routine stores an INTERVAL value in *RESULT*.

Syntax

CALL ECO-DTS USING *DATE1*, *DATE1-LEN*, *DATE1-QUAL*, *DATE2*, *DATE2-LEN*, *DATE2-QUAL*, *RESULT*, *RESULT-LEN*, *RESULT-QUAL*, *STATUS*.

<i>DATE1</i>	the DATETIME value that you provide
<i>DATE1-LEN</i>	the length that you specify for <i>DATE1</i> (INTEGER)
<i>DATE1-QUAL</i>	the qualifier of <i>DATE1</i> (INTEGER) that the ECO-IQU routine provides
<i>DATE2</i>	the DATETIME value that you provide
<i>DATE2-LEN</i>	the length that you specify for <i>DATE2</i> (INTEGER)
<i>DATE2-QUAL</i>	the qualifier of <i>DATE2</i> (INTEGER) that the ECO-IQU routine provides
<i>RESULT</i>	the resultant INTERVAL value that ECO-DTS returns
<i>RESULT-LEN</i>	the length that you specify for <i>RESULT</i> (INTEGER)
<i>RESULT-QUAL</i>	the qualifier of <i>RESULT</i> (INTEGER) that the ECO-IQU routine provides
<i>STATUS</i>	the error status code (INTEGER) that ECO-DTS returns

Usage

Exercise caution when you use the ECO-DTS routine to subtract two DATETIME values. ECO-DTS takes two DATETIME parameters and their corresponding qualifiers and returns a resulting INTERVAL in the format that the INTERVAL qualifier specifies. To get the best results, observe the following guidelines:

- The qualifiers must conform to the SQL standards as specified in the [Informix Guide to SQL: Reference](#) and the [Informix Guide to SQL: Syntax](#).
- The ranges of the two DATETIME qualifiers must overlap.
- The range of the resulting INTERVAL qualifier must overlap the range of the first DATETIME qualifier.
- The result can overflow.

The following two code fragments work correctly, where DATETIME 1 subtracts DATETIME 2.

- The following example does not return an error when the desired INTERVAL result resides in the *month-to-month* or *year-to-month* range:

```
DATETIME 1 = "1995-12"(year-to-month)
DATETIME 2 = "1995-11"(year-to-month)
```

- The following example does not return an error when the desired INTERVAL result resides in the *day-to-hour*, *day-to-minute*, or *day-to-second* range:

```
DATETIME 1 = "12-10 12:20" (month-to-minute)
DATETIME 2 = "12-09 11:10:10" (month-to-second)
```

The next two code fragments return an error, where DATETIME 1 subtracts DATETIME 2.

- The following example returns an error when the desired INTERVAL result spans the *day-to-minute* range because the ranges of DATETIME 1 and DATETIME 2 do not overlap, nor does the range of DATETIME 1 overlap that of the INTERVAL qualifier:

```
DATETIME 1 = "1995-12"(year-to-month)
DATETIME 2 = "10 10:10"(day-to-minute)
```

- The following example returns an error when the desired INTERVAL result spans the *day-to-hour* range because an overflow occurred, so ESQL/COBOL cannot store the result of the operation in the INTERVAL. However, when the desired INTERVAL result spans the *year-to-month* range, an overflow does not occur (and thus, no error).

```
DATETIME 1 = "1995-12-10"(year-to-day)
DATETIME 2 = "1995-12-10 10"(year-to-hour)
```

Return Codes

= 0	Success.
< 0	Failure.

Example

The following code fragment from the ECODTS program tests the ECO-DTS routine. The code fragment accepts two DATETIME qualifiers from the ECO-IQU routine, subtracts the two DATETIME values, and returns a result of type INTERVAL.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECDTS.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 DATE-VAL-1 PIC X(30) VALUE "1998-7".
15 77 DATE-LEN-1 PIC S9(9) COMP-5 VALUE 30.
16 77 DATE-QUAL-1 PIC S9(9) COMP-5.
17 77 DATE-VAL-2 PIC X(30) VALUE "1995-5".
18 77 DATE-LEN-2 PIC S9(9) COMP-5 VALUE 30.
19 77 DATE-QUAL-2 PIC S9(9) COMP-5.
20 77 RESULT PIC X(30).
21 77 RESULT-LEN PIC S9(9) COMP-5 VALUE 30.
22 77 RESULT-QUAL PIC S9(9) COMP-5.
23 77 STAT-CODE PIC S9(9) COMP-5.
24 *
```

```

25  77  QTYPE PIC S9(9) COMP-5 VALUE 0.
26  77  SQUAL PIC X(30) VALUE "year to month".
27  77  SQUAL-LEN PIC S9(9) COMP-5 VALUE 30.
28  77  IQUAL PIC S9(9) COMP-5.
29  77  STAT-CODE-A PIC S9(9) COMP-5.
30  EXEC SQL END DECLARE SECTION END-EXEC.
31
32  PROCEDURE DIVISION.
33  RESIDENT SECTION 1.
34  *****
35  MAIN.
36  *****
37  DISPLAY 'INITIALIZE ECO-IQU.'.
38  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, IQUAL,
39      STAT-CODE-A.
40  DISPLAY IQUAL.
41  MOVE IQUAL TO DATE-QUAL-1.
42  MOVE IQUAL TO DATE-QUAL-2.
43  MOVE IQUAL TO RESULT-QUAL.
44  DISPLAY 'THIS IS A TEST OF ECO-DTS.'.
45  CALL ECO-DTS USING DATE-VAL-1, DATE-LEN-1,
46  DATE-QUAL-1,DATE-VAL-2, DATE-LEN-2, DATE-QUAL-2,
47  RESULT, RESULT-LEN, RESULT-QUAL, STAT-CODE.
48  DISPLAY STAT-CODE.
49  DISPLAY 'DATETIME VALUE 1: ' DATE-VAL-1.
50  DISPLAY 'DATETIME VALUE 2: ' DATE-VAL-2.
51  DISPLAY 'RESULT OF SUBTRACTION: ' RESULT.
52  STOP RUN.

```

Example Output

The output for the preceding code fragment displays the status code, *year-to-month* DATETIME value, DATETIME qualifier type, and INTEGER qualifier for ECO-IQU, plus the status code, first DATETIME value, second DATETIME value, and the result of subtracting the second value from the first with ECO-DTS.

```

INITIALIZE ECO-IQU.
+0000001538
THIS IS A TEST OF ECO-DTS.
+0000000000
DATETIME VALUE 1:  1998-7
DATETIME VALUE 2:  1995-5
RESULT OF SUBTRACTION:      3-02

```

ECO-DTTOASC

Purpose

Use ECO-DTTOASC to convert a string in ANSI DATETIME format to a string in the specified localized format.

Syntax

```
CALL ECO-DTTOASC USING DT, DT-LEN, DT-QUAL, FMTSTR, FMTSTR-LEN,
RES, RES-LEN, STATUS.
```

<i>DT</i>	stores the original DATETIME character string that you provide
<i>DT-LEN</i>	the length that you specify for <i>DT</i> (INTEGER)
<i>DT-QUAL</i>	specifies the INTEGER qualifier for <i>DT</i> that receives its value from the ECO-IQU routine
<i>FMTSTR</i>	the address of the format string for the output (<i>RES</i>), using the directives defined for DBTIME . You provide the value for <i>FMTSTR</i> . When you use the ECO-DTTOASC routine, always specify a format string for <i>FMTSTR</i> . Also, make sure you set the DBTIME environment variable before you use the ECO-DTTOASC routine. For information on how to set DBTIME , refer to the Informix Guide to SQL: Reference .
<i>FMTSTR-LEN</i>	the length that you specify for <i>FMTSTR</i>
<i>RES</i>	stores the resulting formatted character-string representation of the DATETIME value that ECO-DTTOASC
<i>RES-LEN</i>	the length that you specify for <i>RES</i>
<i>STATUS</i>	the error status code that ECO-DTTOASC returns

Usage

The ECO-DTTOASC routine does not require you to make the output qualifier match the input qualifier that the format string specifies. When the output differs from the input qualifier, ECO-DTTOASC performs extensions as shown in the following list:

- Discards fields in *STR* not included in *DT*
- Uses the current time and date to fill in fields to the left of the most-significant field in *STR*
- Uses zeros to fill in fields to the right of the least-significant field in *STR*

If you specify a valid input string and format specification, ECO-DTTOASC sets the output value and returns zero in *STATUS*. Otherwise, ECO-DTTOASC returns an error code and the output string contains unpredictable results.

If *RES* lacks sufficient size to hold the return string, ECO-DTTOASC truncates the return string and returns an error code in *STATUS*.

One of the codes, listed in the following section and returned in the *STATUS* parameter, equals six characters in length (including the minus sign). Thus, you can correctly identify this code only when you define the *STATUS* variable as `S9(x)`, where $x \geq 6$.

When you use a 2-digit year (yy) in a format, the ECO-DTTOASC routine uses the setting of the **DBCENTURY** environment variable to determine the correct century to use. When you do not set **DBCENTURY**, ECO-DTTOASC assumes the 20th century for 2-digit years. For information on how to set **DBCENTURY**, see Chapter 4 of the [Informix Guide to SQL: Reference](#).

GLS

When you use a nondefault locale (one other than U.S. ASCII English) and do not set **DBTIME** or **GL_DATETIME**, the ECO-DTTOASC routine uses the default DATETIME format that the locale defines. For more information, see Chapter 6 of the [Guide to GLS Functionality](#). ♦

Return Codes

-1211	Insufficient memory.
-1260	You cannot convert between the specified types.
-1261	The first field of a DATETIME or INTERVAL value contains too many digits.
-1262	A non-numeric character resides in a DATETIME or INTERVAL value.
-1263	An incorrect or out of range field resides in a DATETIME or INTERVAL value.
-1264	Extra characters exist at the end of a DATETIME or INTERVAL value.
-1265	An overflow occurred on a DATETIME or INTERVAL operation.
-1266	Incompatible INTERVAL or DATETIME values exist.
-1267	A DATETIME computation result exceeds the allowed range.
-1268	Invalid DATETIME qualifier.
-1271	Missing decimal point in fraction.
-1272	You did not specify an input buffer.
-1273	The output buffer either cannot hold the result due to insufficient size or contains a null value.
-1275	Invalid field width for a DATETIME or INTERVAL format string.
-1276	Unsupported format conversion character.
-1277	Input does not match format specification.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

In the following two code fragments, ECO-DTTOASC converts a string in ANSI DATETIME format to a string in the specified format.

```

1 *The input and output qualifiers are the same (hour to second)
2
3 MOVE "01:30:20"                      TO DT.
4 MOVE 8                                TO DT-LEN.
5
6 MOVE 0                                TO FLAG.
7 MOVE "HOUR TO SECOND"                 TO QUAL.
8 MOVE 14                                TO QUAL-LEN.
9
10 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, DT-QUAL, STAT.
11 IF STAT < 0
12     DISPLAY 'DT-QUAL ERROR ', STAT.
13
14 MOVE "%H h %M m %S s"                 TO FMTSTR.
15 MOVE 14                                TO FMTSTR-LEN.
16
17 MOVE 50                                TO RES-LEN.
18
19 *'RES' will be set to "01 h 30 m 20 s"
20 *Use field-width specification to avoid leading zeros
21 *(E.g. %1H).
22
23 CALL ECO-DTTOASC USING DT, DT-LEN, DT-QUAL, FMTSTR,
24     FMTSTR-LEN, RES, RES-LEN, STAT.
25 IF STAT < 0
26     DISPLAY 'DT-TOASC ERROR ', STAT.

```

```

1 *The input and output qualifiers are different
2 *input qual : hour to second
3 *output qual: year to second (ANSI SQL default qualifier)
4
5 MOVE "01:30:20"                      TO DT.
6 MOVE 8                                TO DT-LEN.
7
8 MOVE 0                                TO FLAG.
9 MOVE "HOUR TO SECOND"                 TO QUAL.
10 MOVE 14                                TO QUAL-LEN.
11
12 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, DT-QUAL, STAT.
13 IF STAT < 0
14     DISPLAY 'DT-QUAL ERROR ', STAT.
15
16 *FMTSTR is not initialized.
17 MOVE 50                                TO FMTSTR-LEN.
18
19 MOVE 50                                TO RES-LEN.
20

```

```
21 *'RES' will be set to "XXXX-XX-XX 01:30:20"  
22 *Notice that the output has been extended, and year-month-day  
23 *fields are set to current year, month and day.  
24  
25 CALL ECO-DTTOASC USING DT, DT-LEN, DT-QUAL, FMTSTR,  
26     FMTSTR-LEN, RES, RES-LEN, STAT.  
27 IF STAT < 0  
28     DISPLAY 'DT-TOASC ERROR ', STAT.
```

ECO-DTX

Purpose

Use ECO-DTX to extend a DATETIME value to a different qualifier.

Syntax

CALL ECO-DTX USING *DT*, *DT-LEN*, *DT-QUAL*, *DTRES*, *DTRES-LEN*, *DTRES-QUAL*, *STATUS*.

<i>DT</i>	the DATETIME value that you provide
<i>DT-LEN</i>	the length that you specify for <i>DT</i> (standard INTEGER)
<i>DT-QUAL</i>	the qualifier of <i>DT</i> (standard INTEGER) that the ECO-IQU routine provides
<i>DTRES</i>	the resulting DATETIME value that ECO-DTX returns
<i>DTRES-LEN</i>	the length that you specify for <i>DTRES</i> (standard INTEGER)
<i>DTRES-QUAL</i>	the desired qualifier for <i>DTRES</i> (standard INTEGER) that the ECO-IQU routine provides
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-DTX provides

Usage

The ECO-DTX routine extends the *DT* fields to the qualifier specified in *DTRES-QUAL*.

The ECO-DTX routine discards the *DT* fields that the qualifier specified in *DTRES-QUAL* does not include.

For fields not in *DT*, but specified in *DTRES-QUAL*, ECO-DTX takes the following actions:

- Uses zeros to fill in fields to the right of the least significant field in *DT*
- Uses the current time and date to fill in fields to the left of the most significant field in *DT*

Return Code

-1268 Invalid DATETIME or INTERVAL qualifier.

Example

The following code fragment from the ECODTX program sets a DATETIME value to the date of December 25th for the current year. The ECO-DTX routine generates the current year.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECODTX.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE          PIC S9(9) COMP-5.
18 77 SQUAL          PIC X(30).
19 77 SQUAL-LEN      PIC S9(9) COMP-5 VALUE 30.
20 77 DT             PIC X(30).
21 77 DT-LEN         PIC S9(9) COMP-5 VALUE 30.
22 77 DT-QUAL        PIC S9(9) COMP-5.
23 77 DTRES          PIC X(30).
24 77 DTRES-LEN      PIC S9(9) COMP-5 VALUE 30.
25 77 DTRES-QUAL     PIC S9(9) COMP-5.
26 77 STAT           PIC S9(9) COMP-5 VALUE 30.
27 EXEC SQL END DECLARE SECTION END-EXEC.
28 *
29 PROCEDURE DIVISION.
30 RESIDENT SECTION 1.
31 *
32 *Begin Main routine. Initialize both source and
33 *result qualifiers with ECO-IQU. Extend a DATETIME
34 *value to a different qualifier using ECO-DTX.
35 *Display the resultant value.
36 *
37  MAIN.
```

```

38  MOVE 0 TO QTYPE.
39  MOVE "12-25" TO DT.
40  MOVE "month to day" TO SQUAL.
41  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, DT-QUAL, STAT.
42  DISPLAY ' DT-QUAL = ', DT-QUAL.
43  MOVE "year to hour" TO SQUAL.
44  CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, DTRES-QUAL,
45      STAT.
46  DISPLAY 'RES-QUAL = ', DTRES-QUAL.
47  *
48  DISPLAY 'EXTEND DT FROM "MONTH TO DAY" TO "YEAR TO HOUR"'.
49  CALL ECO-DTX USING DT, DT-LEN, DT-QUAL, DTRES,
50      DTRES-LEN, DTRES-QUAL, STAT.
51  DISPLAY ' DT = ', DT.
52  DISPLAY 'RES = ', DTRES.
53  DISPLAY ' STATUS = ', STAT.
54  STOP RUN.
55  *

```

Example Output

The output for the preceding code fragment displays the DATETIME qualifier and the INTEGER qualifier from ECO-IQU. The ECO-DTX function displays the DATETIME value and the output date (December 25th, 1994).

```

DT-QUAL = +00000001060
RES-QUAL = +00000002566
EXTEND DT FROM "MONTH TO DAY" TO "YEAR TO HOUR"
DT = 12-25
RES = 1994-12-25 00

```

ECO-IDI

Purpose

Use ECO-IDI to divide an INTERVAL value using another INTERVAL value. For example, imagine that you want to find the number of production cycles (a time period or INTERVAL value measures each cycle) for a given time period (an INTERVAL value). To solve that problem, you divide an INTERVAL value using another INTERVAL value to produce a resultant numeric value (number of time periods or intervals).

Syntax

```
CALL ECO-IDI USING INV1, INV1-LEN, INV1-QUAL, INV2, INV2-LEN,  
INV2-QUAL, NUMRES, STATUS.
```

<i>INV1</i>	the INTERVAL value (dividend) that you provide
<i>INV1-LEN</i>	the length that you specify for <i>INV1</i> (standard INTEGER)
<i>INV1-QUAL</i>	the qualifier of <i>INV1</i> (standard INTEGER) that the ECO-IQU routine provides
<i>INV2</i>	the INTERVAL value (divisor) that you provide
<i>INV2-LEN</i>	the length that you specify for <i>INV2</i> (standard INTEGER)
<i>INV2-QUAL</i>	the desired qualifier for <i>INV2</i> (standard INTEGER) the ECO-IQU routine provides
<i>NUMRES</i>	the numeric result (8-byte floating type) that ECO-IDI returns
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-IDI returns

Usage

Make sure you set both the input and output qualifiers to either the *year-to-month* or *day-to-fraction(5)* range.

INV2 divides *INV1* and ECO-IDI stores the result in *NUMRES*. The ECO-IDI routine uses the following formula:

$$INV1/INV2 = NUMRES$$

A positive or negative result can reside in *NUMRES*.

Example

The following code fragment from the ECOIDI program illustrates how an INTERVAL value divides another INTERVAL value.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOIDI.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE          PIC S9(9) COMP-5.
18 77 SQUAL          PIC X(30).
19 77 SQUAL-LEN      PIC S9(9) COMP-5 VALUE 30.
20 77 INV1           PIC X(30).
21 77 INV1-LEN       PIC S9(9) COMP-5 VALUE 30.
22 77 INV1-QUAL      PIC S9(9) COMP-5.
23 77 INV2           PIC X(30).
24 77 INV2-LEN       PIC S9(9) COMP-5 VALUE 30.
25 77 INV2-QUAL      PIC S9(9) COMP-5.
26 77 NUMRES         PIC S9(10)V9(8) COMP-3.
27 77 STAT           PIC S9(9) COMP-5 VALUE 30.
28 EXEC SQL END DECLARE SECTION END-EXEC.
29 *
```

```

30 77 DISPRES          PIC +9(10).9(8) USAGE DISPLAY.
31 *
32 PROCEDURE DIVISION.
33 RESIDENT SECTION 1.
34 *
35 *Begin Main routine. Initialize both source and
36 *result qualifiers with the ECO-IQU routine. Divide
37 *an INTERVAL value with another INTERVAL value using
38 *the ECO-IDI routine. Display the resultant value.
39 *
40 MAIN.
41 MOVE 1 TO QTYPE.
42 MOVE "hour to minute" TO SQUAL.
43 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN,
44      INV1-QUAL, STAT.
45 DISPLAY 'INV1-QUAL = ', INV1-QUAL.
46 MOVE "hour to minute" TO SQUAL.
47 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN,
48      INV2-QUAL, STAT.
49 DISPLAY 'INV2-QUAL = ', INV2-QUAL.
50 *
51 MOVE "75:27" TO INV1.
52 MOVE "19:10" TO INV2.
53 DISPLAY 'DIVIDE "HOUR TO MINUTE"...'.
54 DISPLAY 'BY "HOUR TO MINUTE"'.
55 CALL ECO-IDI USING INV1, INV1-LEN, INV1-QUAL,
56      INV2, INV2-LEN, INV2-QUAL, NUMRES, STAT.
57 DISPLAY 'STATUS = ', STAT.
58 DISPLAY 'INV1 = ', INV1.
59 DISPLAY 'INV2 = ', INV2.
60 MOVE NUMRES TO DISPRES.
61 DISPLAY 'RES = ', DISPRES..
62 STOP RUN.
63 *

```

Example Output

The output for the preceding code fragment displays the qualifiers of the dividend and divisors, the status code that ECO-IDI returns, the dividend and divisor, and the unformatted and formatted result. The following output tells you that three of the specified *hour-to-minute* INTERVAL units reside in the specified *hour-to-minute* INTERVAL value.

```
INV1-QUAL = +0000001128
INV2-QUAL = +0000001128
DIVIDE "HOUR TO MINUTE"...
BY INV2 "HOUR TO MINUTE"
STATUS = +0000000000
INV1 = 75:27
INV2 = 19:10
RES = +0000000003.93652174
```

ECO-IDN

Purpose

Use ECO-IDN to divide an INTERVAL value using a numeric value. For example, imagine that you already specified a given time period for production (an INTERVAL value) and you knew how many production cycles (a numeric value) you wanted to run in that time period, but you did not know the length and type of each cycle (the resulting interval value). To make that issue possible, use a numeric value to divide the interval value to produce a resultant interval value.

Syntax

```
CALL ECO-IDN USING INV, INV-LEN, INV-QUAL, NUM, INVRES, INVRES-LEN, INVRES-QUAL, STATUS.
```

<i>INV</i>	the INTERVAL value that you provide
<i>INV-LEN</i>	the length that you specify for <i>INV</i> (standard INTEGER)
<i>INV-QUAL</i>	the qualifier of <i>INV</i> (standard INTEGER) that the ECO-IQU routine provides
<i>NUM</i>	the numeric value (8-byte floating type) that you provide
<i>INVRES</i>	the resulting INTERVAL value that ECO-IDN returns
<i>INVRES-LEN</i>	the length that you specify for <i>INVRES</i> (standard INTEGER)
<i>INVRES-QUAL</i>	the desired qualifier for <i>INVRES</i> (standard INTEGER) that the ECO-IQU routine provides
<i>STATUS</i>	the status error code (standard INTEGER) that ECO-IDN returns

Usage

Make sure you set both the input and output qualifiers to either the *year-to-month* or *day-to-fraction(5)* range.

NUM divides *INV* and ECO-IDN stores the result in *INVRES*. The ECO-IDN routine uses the following formula to determine the resulting value:

$$INV / NUM = INVRES$$

A positive or negative value can reside in *NUM*.

If the *INVRES* qualifier differs from the *INV* qualifier, ECO-IDN extends the result as defined within the ECO-INX routine.

Example

The following code fragment from the ECOIDN program illustrates dividing an INTERVAL value using a numeric value using the ECO-IDN routine. It illustrates the result of INTERVAL division when the input and output qualifiers differ.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOIDN.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
```



```

16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE          PIC S9(9) COMP-5.
18 77 SQUAL          PIC X(30).
19 77 SQUAL-LEN      PIC S9(9) COMP-5 VALUE 30.
20 77 INV            PIC X(30).
21 77 INV-LEN        PIC S9(9) COMP-5 VALUE 30.
22 77 INV-QUAL       PIC S9(9) COMP-5.
23 77 INVRES         PIC X(30).
24 77 INVRES-LEN     PIC S9(9) COMP-5 VALUE 30.
25 77 INVRES-QUAL    PIC S9(9) COMP-5.
26 77 NUM            PIC S9(10)V9(8) COMP-3.
27 77 STAT           PIC S9(9) COMP-5 VALUE 30.
28 EXEC SQL END DECLARE SECTION END-EXEC.
29 *
30 PROCEDURE DIVISION.
31 RESIDENT SECTION 1.
32 *
33 *Begin Main routine. Initialize both source and result
34 *qualifiers by using the ECO-IQU routine. Divide the INTERVAL
35 *value by a numeric value using the ECO-IDN routine. Display
36 *the resultant value.
37 *
38 MAIN.
39 MOVE 1 TO QTYPE.
40 MOVE "hour to minute" TO SQUAL.
41 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, INV-QUAL, STAT.
42 DISPLAY 'INV-QUAL = ', INV-QUAL.
43 MOVE "hour to minute" TO SQUAL.
44 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN,
45 INVRES-QUAL, STAT.
46 DISPLAY 'RES-QUAL = ', INVRES-QUAL.
47 *
48 MOVE "25:15" TO INV.
49 MOVE 5.00 TO NUM.
50 DISPLAY 'DIVIDE INV "HOUR TO MINUTE" ', INV.
51 DISPLAY 'BY ', NUM.
52 CALL ECO-IDN USING INV, INV-LEN, INV-QUAL, NUM,
53 INVRES, INVRES-LEN, INVRES-QUAL, STAT.
54 DISPLAY 'STATUS = ', STAT.
55 DISPLAY 'INV = ', INV.
56 DISPLAY 'RES = ', INVRES.
57 STOP RUN.
58 *

```

Example Output

The output for the preceding code fragment displays the two INTERVAL qualifiers, the status code, the INTERVAL value divided by the number 5.00, and the resulting INTERVAL value.

```
INV-QUAL = +0000001128
RES-QUAL = +0000001128
DIVIDE INV "HOUR TO MINUTE" 25:15
BY +000000000500000000
STAT = +0000000000
INV = 25:15
RES = 5:03
```

ECO-IMN

Purpose

Use ECO-IMN to multiply an INTERVAL value by a numeric value.

Syntax

```
CALL ECO-IMN USING INV, INV-LEN, INV-QUAL, NUM, INVRES, INVRES-LEN, INVRES-QUAL, STATUS.
```

<i>INV</i>	the INTERVAL value that you provide
<i>INV-LEN</i>	the length that you provide for <i>INV</i> (standard INTEGER)
<i>INV-QUAL</i>	the qualifier of <i>INV</i> (standard INTEGER) that the ECO-IQU routine provides
<i>NUM</i>	the numeric value (8-byte floating type) that you provide
<i>INVRES</i>	the resulting INTERVAL value that ECO-IMN returns
<i>INVRES-LEN</i>	the length that you specify for <i>INVRES</i> (standard INTEGER)
<i>INVRES-QUAL</i>	the desired qualifier for <i>INVRES</i> (standard INTEGER) that the ECO-IQU routine provides
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-IMN returns

Usage

The ECO-IMN routine multiplies *INV* by *NUM* and stores the result in *INVRES*. The ECO-IMN routine uses the following formula to determine the resulting value:

$$INV * NUM = INVRES$$

Make sure you set the input and output qualifiers to the *year-to-month* or *day-to-fraction(5)* range.

NUM multiplies *INV* and ECO-IMN stores the result in *INVRES*. A positive or a negative value can reside in *NUM*.

If the qualifier for *INVRES* differs from the qualifier for *INV*, ECO-IMN extends the result as defined in the ECO-INX routine.

Example

The following code fragment from the ECOIMN program illustrates multiplying an INTERVAL value by a numeric value using the ECO-IMN routine. The code fragment illustrates the result of INTERVAL multiplication when the input and output qualifiers differ.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOIMN.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE          PIC S9(9) COMP-5.
18 77 SQUAL          PIC X(30).
19 77 SQUAL-LEN      PIC S9(9) COMP-5 VALUE 30.
20 77 INV            PIC X(30).
21 77 INV-LEN        PIC S9(9) COMP-5 VALUE 30.
22 77 INV-QUAL       PIC S9(9) COMP-5.
23 77 INVRES         PIC X(30).
24 77 INVRES-LEN     PIC S9(9) COMP-5 VALUE 30.
25 77 INVRES-QUAL    PIC S9(9) COMP-5.
26 77 NUM            PIC S9(10)V9(8) COMP-3.
27 77 STAT           PIC S9(9) COMP-5 VALUE 30.
28 EXEC SQL END DECLARE SECTION END-EXEC.
29 *
30 PROCEDURE DIVISION.
31 RESIDENT SECTION 1.
32 *
33 *Begin Main routine. Initialize both source and
34 *result qualifiers using ECO-IQU. Multiply an
35 *INTERVAL value by a numeric value using the ECO-IMN
36 *routine. This generates a resultant value.
37 *
38 MAIN.
39 MOVE 1 TO QTYPE.
40 MOVE "hour to minute" TO SQUAL.
41 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN,

```

```

42     INV-QUAL, STAT.
43     DISPLAY 'INV-QUAL = ', INV-QUAL.
44     MOVE "hour to minute" TO SQUAL.
45     CALL ECO-IQU USING QTYPE, SQUAL,
46         SQUAL-LEN, INVRES-QUAL, STAT.
47     DISPLAY 'RES-QUAL = ', INVRES-QUAL.
48     *
49     MOVE "12:15" TO INV.
50     MOVE 5.0 TO NUM.
51     DISPLAY 'MULTIPLY INV "HOUR TO MINUTE" ', INV, 'BY ', NUM.
52     CALL ECO-IMN USING INV, INV-LEN, INV-QUAL, NUM, INVRES,
53         INVRES-LEN, INVRES-QUAL, STAT.
54     DISPLAY 'STATUS = ', STAT.
55     DISPLAY 'INV = ', INV.
56     DISPLAY 'RES = ', INVRES.
57     DISPLAY ' '.
58     STOP RUN.
59     *

```

Example Output

The output for the preceding code fragment displays the INTERVAL qualifier, the numeric value used to multiply the INTERVAL value, the status code, the INTERVAL value, and the resulting INTERVAL value after multiplication.

```

INV-QUAL = +00000001128
RES-QUAL = +00000001128
MULTIPLY INV "HOUR TO MINUTE" 12:15 BY +000000000500000000
STAT = +00000000000
INV = 12:15
RES = 61:15

```

ECO-INCVASC

Purpose

Use ECO-INCVASC to convert a string with a specified format to an ANSI INTERVAL string.

Syntax

CALL ECO-INCVASC USING *STR*, *STR-LEN*, *FMTSTR*, *FMTSTR-LEN*, *INTRVL*, *INTRVL-LEN*, *INTRVL-QUAL*, *STATUS*.

<i>STR</i>	the address of the input INTERVAL string in the format of <i>FMTSTR</i> that you provide
<i>STR-LEN</i>	the length that you specify for <i>STR</i>
<i>FMTSTR</i>	the address of the format string for the input (<i>STR</i>) that you provide
<i>FMTSTR-LEN</i>	the length that you specify for <i>FMTSTR</i>
<i>INTRVL</i>	the address of the resulting ANSI INTERVAL string that ECO-INCVASC returns
<i>INTRVL-LEN</i>	the length that you specify for <i>INTRVL</i>
<i>INTRVL-QUAL</i>	the qualifier for <i>INTRVL</i> that the ECO-IQU routine provides
<i>STATUS</i>	the error status code that ECO-INCVASC returns

Usage

The input string can contain leading and trailing spaces. However, from the first to last significant digit, ECO-INCVASC accepts only digits and delimiters appropriate to the fields that the formatted string implies.

When you call ECO-INCVASC, make sure you use only contiguous fields in the INTERVAL input string. If, for example, you specify the qualifier as *hour-to-second*, you must make sure that the values for hour, minute, and second reside in the string (not necessarily in that order). Otherwise, you receive an error.

ECO-INCVASC does not require you to make the output qualifier match the input qualifier as the format string specifies. When the output qualifier differs from the input qualifier, ECO-INCVASC converts the result to appropriate units. However, both the input and the output must represent an interval with a span of *year-to-month* or *day-to-fraction*.

If you provide a valid input string and format specification, ECO-INCVASC sets the output value and returns zero in *STATUS*. Otherwise, ECO-INCVASC returns an error code and the output string contains unpredictable results.

If you use an empty *FMTSTR* argument, ECO-INCVASC returns an error. Figure 3-8 shows the formatting directives you can use in *FMTSTR*.

Figure 3-8
Formatting Directives for FMTSTR

String	Use
%d	replaced with the day of the month as a decimal number [01,31].
%Fn	replaced with the value of the fraction with precision that the integer <i>n</i> specifies. The default value of <i>n</i> equals 2; the range of <i>n</i> equals 0 <= <i>n</i> <= 5.
%H	replaced with the hour (24-hour clock) as a decimal number [00,23].
%I	replaced with the hour (12-hour clock) as a decimal number [01,12].
%M	replaced with the minute as a decimal number [00,59].
%m	replaced with the month as a decimal number [01,12].
%S	replaced with the second as a decimal number [00,59].
%y	replaced with the year as a 2-digit decimal number [00,99]. You must interpret the format for an INTERVAL value literally: “88” means “0088,” not “1988.”
%Y	replaced with the year as a 4-digit decimal number; use Y for an interval of more than 99 years.
%%	replaced with % (to allow % in the format string).

If *INTRVL* lacks sufficient size, ECO-INCVASC truncates the result and returns an error.

One of the codes, listed in the following section and returned in the *STATUS* parameter, equals six characters in length (including the minus sign). Thus, you can correctly identify this code only when you define the *STATUS* variable as *S9(x)*, where $x \geq 6$.

Return Codes

-1211	Insufficient memory.
-1260	You cannot convert between the specified types.
-1261	The first field of a DATETIME or INTERVAL value contains too many digits.
-1262	Non-numeric character resides in a DATETIME or INTERVAL value.
-1263	An incorrect or out of range field resides in a DATETIME or INTERVAL value.
-1264	Extra characters exist at the end of a DATETIME or INTERVAL value.
-1265	An overflow occurred on a DATETIME or INTERVAL operation.
-1266	Incompatible DATETIME or INTERVAL values exist.
-1267	A DATETIME computation result exceeds the allowed range.
-1268	Invalid DATETIME qualifier.
-1271	Missing decimal point in fraction.
-1272	You did not specify an input buffer.
-1273	The output buffer either cannot hold the result due to insufficient size or contains a null value.
-1275	Invalid field width for a DATETIME or INTERVAL format string.
-1276	Unsupported format conversion character.
-1277	Input does not match format specification.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The two following code fragments convert a string with a specified format to an ANSI INTERVAL string.

```

1  *The input and output qualifiers are the same
2  *(day to minute).
3
4  MOVE "20 days, 3 hours, 40 minutes"      TO STR.
5  MOVE 28                                TO STR-LEN.
6
7  *Note the absence of field-width and precision
8  *specification in the input format string.
9  MOVE "%d days, %H hours, %M minutes"     TO FMTSTR.
10 MOVE 29                                TO FMTSTR-LEN.
11
12 MOVE 1                                  TO FLAG.
13 MOVE "DAY TO MINUTE"                    TO QUAL.
14 MOVE 13                                TO QUAL-LEN.
15
16 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, INTVL-QUAL, STAT.
17 IF STAT < 0
18 DISPLAY 'INTVL-QUAL ERROR ', STAT.
19
20 MOVE 50                                TO INTVL-LEN.
21
22 *'INTVL' will be set to "20 03:40"
23
24 CALL ECO-INCVASC USING STR, STR-LEN, FMTSTR, FMTSTR-LEN,
25 INTVL, INTVL-LEN, INTVL-QUAL, STAT.
26 IF STAT < 0
27 DISPLAY 'IN-CVASC ERROR ', STAT.
28 *The input and output qualifiers are different
29 *input qual : day to minute
30 *output qual: hour to second
31
32 MOVE "20 days, 3 hours, 40 minutes"      TO STR.
33 MOVE 28                                TO STR-LEN.
34
35 MOVE "%d days, %H hours, %M minutes"     TO FMTSTR.
36 MOVE 29                                TO FMTSTR-LEN.
37
38 MOVE 1                                  TO FLAG.
39
40 *Since the expected number of digits in hours is more than 2,
41 *set the value to some maximum [ HOUR(5) ].
42 MOVE "HOUR(5) TO SECOND"                 TO QUAL.
43 MOVE 17                                TO QUAL-LEN.

```

```
44
45 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, INTVL-QUAL, STAT.
46 IF STAT < 0
47     DISPLAY 'INTVL-QUAL ERROR ', STAT.
48
49 MOVE 50                                TO INTVL-LEN.
50
51 *'INTVL' will be set to "483:40:00"
52 *Notice that "20 days and 3 hours" have become "483 hours"
53 *and seconds field has been set to "00".
54
55 CALL ECO-INCVASC USING STR, STR-LEN, FMTSTR, FMTSTR-LEN,
56     INTVL, INTVL-LEN, INTVL-QUAL, STAT.
57 IF STAT < 0
58     DISPLAY 'IN-CVASC ERROR ', STAT.
```

ECO-INTOASC

Purpose

Use ECO-INTOASC to convert a string in ANSI INTERVAL format to an ASCII string in the specified localized format.

Syntax

```
CALL ECO-INTOASC USING INTRVL, INTRVL-LEN, INTRVL-QUAL, FMTSTR,
FMTSTR-LEN, RES, RES-LEN, STATUS.
```

<i>INTRVL</i>	stores the original INTERVAL character string that you provide
<i>INTRVL-LEN</i>	the length that you specify for <i>INTRVL</i> (INTEGER)
<i>INTRVL-QUAL</i>	specifies the INTEGER qualifier for <i>INTRVL</i> that the ECO-IQU routine provides
<i>FMTSTR</i>	the address of the format string for the output (<i>RES</i>) that you provide
<i>FMTSTR-LEN</i>	the length that you specify for <i>FMTSTR</i>
<i>RES</i>	stores the resulting formatted character-string representation of the INTERVAL value that ECO-INTOASC returns
<i>RES-LEN</i>	the length that you specify for <i>RES</i>
<i>STATUS</i>	the error status code that ECO-INTOASC returns

Usage

ECO-INTOASC does not require that you make the output qualifier match the input qualifier as the format string specifies. When the output qualifier differs from the input identifier, ECO-INTOASC converts the result to appropriate units. However, both the input and the output qualifiers must represent an interval with a span of *year-to-month* or *day-to-fraction*.

If you provide a valid input string and format specification, ECO-INTOASC sets the output value and returns zero in *STATUS*. Otherwise, ECO-INTOASC returns an error code and the output string contains unpredictable results.

If you use an empty *FMTSTR* argument, ECO-INTOASC returns an error. Figure 3-9 shows the formatting directives you can use in *FMTSTR*:

Figure 3-9
Formatting Directives for FMTSTR

String	Use
%d	replaced with the day of the month as a decimal number [01,31].
%Fn	replaced with the value of the fraction with precision that the integer <i>n</i> specifies. The default value of <i>n</i> equals 2; the range of <i>n</i> equals 0 <= <i>n</i> <= 5.
%H	replaced with the hour (24-hour clock) as a decimal number [00,23].
%I	replaced with the hour (12-hour clock) as a decimal number [01,12].
%M	replaced with the minute as a decimal number [00,59].
%m	replaced with the month as a decimal number [01,12].
%S	replaced with the second as a decimal number [00,59].
%y	replaced with the year as a 2-digit decimal number [00,99]. You must interpret the format for an INTERVAL value literally: “88” means “0088,” not “1988.”
%Y	replaced with the year as a 4-digit decimal number; use Y for an interval of more than 99 years.
%%	replaced with % (to allow % in the format string).

Use the %Y directive when the interval exceeds 99 years because %y can handle only two digits. Use %H for hours, not %I, because %I can handle only 12 hours.

If *RES* lacks sufficient size to hold the return string, ECO-INTOASC truncates the return string and returns an error code in *STATUS*.

One of the codes, listed in the following section and returned in the *STATUS* parameter, equals six characters in length (including the minus sign). Thus, you can correctly identify this code only when you define the *STATUS* variable as *S9(x)*, where *x* >= 6.

Return Codes

-1211	Insufficient memory.
-1260	You cannot convert between the specified types.
-1261	The first field of a DATETIME or INTERVAL value contains too many digits.
-1262	A non-numeric character resides in a DATETIME or INTERVAL value.
-1263	An incorrect or out of range field resides in a DATETIME or INTERVAL value.
-1264	Extra characters exist at the end of a DATETIME or INTERVAL value.
-1265	An overflow occurred on a DATETIME or INTERVAL operation.
-1266	Incompatible DATETIME or INTERVAL values exist.
-1267	A DATETIME computation result exceeds the allowed range.
-1268	Invalid DATETIME qualifier.
-1271	Missing decimal point in fraction.
-1272	You did not specify an input buffer.
-1273	The output buffer either cannot hold the result due to insufficient size or contains a null value.
-1275	Invalid field width for a DATETIME or INTERVAL format string.
-1276	Unsupported format-conversion character.
-1277	Input does not match format specification.
-22275	INTERNAL ERROR: You exceeded the temporary buffer length.

Example

The following two code fragments convert a string in ANSI INTERVAL format to an ASCII string in the specified format.

```

1  *The input and output qualifiers are the same
2  *(day to minute)
3
4  MOVE "20 03:40"                                TO INTVL.
5  MOVE 8                                           TO INTVL-LEN.
6
7  MOVE 1                                           TO FLAG.
8  MOVE "DAY TO MINUTE"                          TO QUAL.
9  MOVE 13                                         TO QUAL-LEN.
10
11 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, INTVL-QUAL, STAT.
12 IF STAT < 0
13     DISPLAY 'INTVL-QUAL ERROR ', STAT.
14
15 MOVE "%1d days, %1H hours and %1M minutes to go" TO FMTSTR.
16 MOVE 41                                           TO FMTSTR-LEN.
17
18 MOVE 50                                           TO RES-LEN.
19
20 *'RES' will be set to "20 days, 3 hours and 40 minutes to go"
21
22 CALL ECO-INTOASC USING INTVL, INTVL-LEN, INTVL-QUAL,
23     FMTSTR, FMTSTR-LEN, RES, RES-LEN, STAT.
24 IF STAT < 0
25     DISPLAY 'IN-CVASC ERROR ', STAT.
26 *The input and output qualifiers are different
27 *input qual : day to minute
28 *output qual: hour to second
29
30 MOVE "20 03:40"                                TO INTVL.
31 MOVE 8                                           TO INTVL-LEN.
32 MOVE 1                                           TO FLAG.
33 MOVE "DAY TO MINUTE"                          TO QUAL.
34 MOVE 13                                         TO QUAL-LEN.
35
36 CALL ECO-IQU USING FLAG, QUAL, QUAL-LEN, INTVL-QUAL, STAT.
37 IF STAT < 0
38     DISPLAY 'INTVL-QUAL ERROR ', STAT.
39
40 MOVE "%1H hours, %1M minutes and %1S seconds to go" TO
41     FMTSTR.
42 MOVE 44                                           TO FMTSTR-LEN.
43 MOVE 50                                           TO RES-LEN.

```

```
44 *'RES' will be set to "483 hours 40 minutes and 0 seconds to
45   go"
46 *Notice that "20 days and 3 hours" have become "483 hours" and
47 *the seconds field has been set to zero.
48
49 CALL ECO-INTOASC USING INTVL, INTVL-LEN, INTVL-QUAL,
50     FMTSTR, FMTSTR-LEN, RES, RES-LEN, STAT.
51 IF STAT < 0
52     DISPLAY 'IN-CVASC ERROR ', STAT.
```


ECO-INX

Purpose

Use ECO-INX to extend an INTERVAL value to a different qualifier.

Syntax

```
CALL ECO-INX USING INV, INV-LEN, INV-QUAL, INVRES, INVRES-LEN,
INVRES-QUAL, STATUS.
```

<i>INV</i>	the INTERVAL value that you provide
<i>INV-LEN</i>	the length that you specify for <i>INV</i> (standard INTEGER)
<i>INV-QUAL</i>	the qualifier of <i>INV</i> (standard INTEGER) that the ECO-IQU routine provides
<i>INVRES</i>	the resulting INTERVAL value that ECO-INX returns
<i>INVRES-LEN</i>	the length that you specify for <i>INVRES</i> (standard INTEGER)
<i>INVRES-QUAL</i>	the desired qualifier for <i>INVRES</i> (standard INTEGER) that the ECO-IQU routine provides
<i>STATUS</i>	the error status code (standard INTEGER) that ECO-INX returns

Usage

Make sure you set both the input and output qualifiers to the *year-to-month* or *day-to-fraction(5)* range.

The ECO-INX routine copies the *INV* fields to *INVRES*, with the copy *INVRES-QUAL* controls.

The ECO-INX routine discards the *INV* fields that reside to the right of the least significant field in *INVRES*.

For fields not in *INV*, but specified in *INVRES-QUAL*, the ECO-INX routine takes the following actions:

- Uses zeros to fill in fields to the right of the least significant digit
- Uses valid *INTERVAL* values to fill in fields to the left of the most-significant field in *INV*

Example

The following code fragment from the ECOINX program extends an *INTERVAL* value to a different qualifier using the ECO-INX routine. Note that the output contains zeros in the *seconds* field, and the *days* field contains the number 3.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOINX.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE                PIC S9(9) COMP-5.
18 77 SQUAL                PIC X(30).
19 77 SQUAL-LEN            PIC S9(9) COMP-5 VALUE 30.
20 77 INV                  PIC X(30).
21 77 INV-LEN              PIC S9(9) COMP-5 VALUE 30.
22 77 INV-QUAL             PIC S9(9) COMP-5.
23 77 INVRES               PIC X(30).
24 77 INVRES-LEN           PIC S9(9) COMP-5 VALUE 30.
25 77 INVRES-QUAL          PIC S9(9) COMP-5.
26 77 STAT                 PIC S9(9) COMP-5 VALUE 30.
27 EXEC SQL END DECLARE SECTION END-EXEC.
28 *
29 PROCEDURE DIVISION.
30 RESIDENT SECTION 1.
31 *
```

```

32 *Begin Main routine. Initialize both source and
33 *result qualifiers using the ECO-IQU routine. Extend
34 *an INTERVAL value using the ECO-INX routine. This
35 *creates a resultant value.
36 *
37 MAIN.
38 MOVE 1 TO QTYPE.
39 MOVE "hour to minute" TO SQUAL.
40 CALL ECO-IQU USING QTYPE, SQUAL,
41     SQUAL-LEN, INV-QUAL, STAT.
42 DISPLAY 'INV-QUAL = ', INV-QUAL.
43 MOVE "day to second" TO SQUAL.
44 CALL ECO-IQU USING QTYPE, SQUAL,
45     SQUAL-LEN, INVRES-QUAL, STAT.
46 DISPLAY 'RES-QUAL = ', INVRES-QUAL.
47 *
48 MOVE "75:27" TO INV.
49 DISPLAY 'EXTEND INV FROM "HOUR TO MINUTE" TO
50 "DAY TO SECOND"'.
51 CALL ECO-INX USING INV, INV-LEN, INV-QUAL, INVRES,
52     INVRES-LEN, INVRES-QUAL, STAT.
53 DISPLAY 'INV = ', INV.
54 DISPLAY 'RES = ', INVRES.
55 DISPLAY ' '.
56 STOP RUN.
57 *

```

Example Output

The output for the preceding code fragment displays two INTEGER qualifiers from ECO-IQU, and the input and output INTERVAL values from ECO-INX.

```

INV-QUAL = +0000001128
RES-QUAL = +0000002122
EXTEND INV FROM "HOUR TO MINUTE" TO "DAY TO SECOND"
INV = 75:27
RES = 3 03:27:00

```

ECO-IQU

Purpose

Use ECO-IQU to determine the INTEGER qualifier for a given character-string qualifier.

Syntax

CALL ECO-IQU USING *QTYPE*, *SQUAL*, *SQUAL-LEN*, *IQUAL*, *STATUS*.

<i>QTYPE</i>	the qualifier type (INTEGER) that you provide: 0 = DATETIME 1 = INTERVAL
<i>SQUAL</i>	the character-string qualifier that you provide
<i>SQUAL-LEN</i>	the length that you specify for <i>SQUAL</i> (INTEGER)
<i>IQUAL</i>	the INTEGER qualifier that ECO-IQU returns
<i>STATUS</i>	the error status code (INTEGER) that ECO-IQU returns

Return Codes

0	Success.
-1261	The first field of a DATETIME or INTERVAL value contains too many digits.
-1262	A nonnumeric character in DATETIME or INTERVAL value.
-1263	An out of range field resides in a DATETIME or INTERVAL value.
-1264	Extra characters exist at the end of a DATETIME or INTERVAL value.
-1268	Invalid DATETIME qualifier.

Example

This following code fragment from the ECOIQU program determines the INTEGER qualifier for a character string for both a DATETIME and an INTERVAL data type.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOIQU.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
14 77 QTYPE PIC S9(9) COMP-5 VALUE 0.
15 77 SQUAL PIC X(30) VALUE "year to month".
16 77 SQUAL-LEN PIC S9(9) COMP-5 VALUE 30.
17 77 IQUAL PIC S9(9) COMP-5.
18 77 STAT-CODE-A PIC S9(9) COMP-5.
19 *
20 77 ZQUAL PIC S9(9) COMP-5.
21 *
22 EXEC SQL END DECLARE SECTION END-EXEC.
23
24
25 PROCEDURE DIVISION.
26 RESIDENT SECTION 1.
27 *****
28 MAIN.
29 *****
30 DISPLAY 'THIS IS A TEST OF ECO-IQU.'.
31 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, IQUAL,
32     STAT-CODE-A.
33 DISPLAY SQUAL.
34 DISPLAY 'DATETIME QUALIFIER VALUE: ' IQUAL.
35 MOVE 1 TO QTYPE.
36 MOVE "year to month" TO SQUAL.
37 CALL ECO-IQU USING QTYPE, SQUAL, SQUAL-LEN, ZQUAL,
38     STAT-CODE-A.
39 DISPLAY SQUAL.
40 DISPLAY 'INTERVAL QUALIFIER VALUE: ' ZQUAL.
41 DISPLAY ' '.
42 STOP RUN.

```

Example Output

The output for the preceding code fragment displays the *year-to-month* qualifier, DATETIME INTEGER qualifier value, *year-to-month* qualifier, and INTERVAL INTEGER qualifier value for ECO-IQU.

```
THIS IS A TEST OF ECO-IQU.  
year to month  
DATETIME QUALIFIER VALUE:  +0000001538  
year to month  
INTERVAL QUALIFIER VALUE:  +0000001538
```

ECO-SQU

Purpose

Use ECO-SQU to determine the character-string qualifier for a given INTEGER qualifier.

Syntax

CALL ECO-SQU USING <i>QTYPE</i> , <i>IQUAL</i> , <i>SQUAL</i> , <i>SQUAL-LEN</i> , <i>STATUS</i> .	
<i>QTYPE</i>	the qualifier type (INTEGER) that you provide: 0 = DATETIME 1 = INTERVAL
<i>IQUAL</i>	the resultant INTEGER qualifier that ECO-SQU returns
<i>SQUAL</i>	the character-string qualifier that the ECO-IQU routine provides
<i>SQUAL-LEN</i>	the length that you specify for <i>SQUAL</i> (INTEGER)
<i>STATUS</i>	the error status code (INTEGER) that ECO-SQU returns

Return Codes

= 0	Success.
< 0	Failure.

Example

The following code fragment from the ECOSQU program tests the ECO-SQU routine and determines the character-string qualifier when given an integer qualifier from the routine ECO-IQU.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOSQU.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 QTYPE PIC S9(9) COMP-5 VALUE 1.
18 77 SQUAL PIC X(30).
19 77 SQUAL-LEN PIC S9(9) COMP-5 VALUE 30.
20 77 IQUAL PIC S9(9) COMP-5.
21 77 STAT-CODE PIC S9(9) COMP-5.
22 77 QTYPE-A PIC S9(9) COMP-5 VALUE 1.
23 77 SQUAL-A PIC X(30) VALUE "year to month".
24 77 SQUAL-LEN-A PIC S9(9) COMP-5 VALUE 30.
25 77 IQUAL-A PIC S9(9) COMP-5.
26 77 STAT-CODE-A PIC S9(9) COMP-5.
27 EXEC SQL END DECLARE SECTION END-EXEC.
28 *
29 PROCEDURE DIVISION.
30 RESIDENT SECTION 1.
31 *
32 *Begin Main routine. Use the ECO-IQU routine to initialize
33 *an integer value. Use the ECO-SQU routine to determine the
34 *character string qualifier associated with the integer
35 *qualifier returned by ECO-IQU.
36 *
37 MAIN.
38 CALL ECO-IQU USING QTYPE-A, SQUAL-A, SQUAL-LEN-A,
39 IQUAL-A, STAT-CODE-A.
40 DISPLAY STAT-CODE-A.
41 DISPLAY SQUAL-A.
42 DISPLAY QTYPE-A.

```



```

42  DISPLAY IQUAL-A.
43  *
44  DISPLAY 'THIS IS A TEST OF ECO-SQU.'.
45  MOVE IQUAL-A TO IQUAL.
46  CALL ECO-SQU USING QTYPE, IQUAL, SQUAL,
47  SQUAL-LEN, STAT-CODE.
48  DISPLAY STAT-CODE.
49  DISPLAY SQUAL.
50  DISPLAY QTYPE.
51  DISPLAY IQUAL.
52  STOP RUN.
53  *

```

Example Output

The output for the preceding code fragment displays the status, *year-to-month* qualifier, INTERVAL qualifier type, and INTERVAL INTEGER qualifier for ECO-IQU, plus the status, *year-to-month* qualifier, INTERVAL qualifier type, and INTERVAL INTEGER qualifier for ECO-SQU.

```

INITIALIZE INTEGER VALUE IN ECO-IQU.
+0000000000
year to month
+0000000001
+0000001538
THIS IS A TEST OF ECO-SQU.
+0000000000
year to month
+0000000001
+0000001538

```


Error Handling

Obtaining Diagnostic Information After an SQL Statement Executes.	4-4
The GET DIAGNOSTICS Statement	4-4
Statement Information	4-4
Exception Information	4-5
Examples Illustrating the GET DIAGNOSTICS Statement	4-7
Using the SQLSTATE Variable	4-9
Class and Subclass Codes	4-9
List of SQLSTATE Codes	4-11
Using SQLSTATE in Applications	4-14
Multiple Error Conditions	4-16
The SQLCA Record.	4-17
The Contents of the SQLCA Structure	4-19
Using SQLCODE OF SQLCA	4-21
Codes for SQL Statement Results	4-21
Success	4-22
Success with Warning	4-22
No Data Found.	4-22
SQLSTATE Class Code = 02	4-22
SQLCODE OF SQLCA = 100	4-23
Error	4-24
Errors After a PREPARE Statement	4-24
Errors After an EXECUTE Statement	4-24
When an Error Occurs on GET DIAGNOSTICS	4-25

Error Handling in Programs	4-25
Checking for Errors with the GET DIAGNOSTICS Statement	4-25
Checking for an Error Using In-Line Code	4-26
Automatically Checking for Errors Using the WHENEVER Statement	4-29
Checking for Warnings Using GET DIAGNOSTICS	4-35
Checking for Warnings Using the SQLWARN OF SQLCA Structure	4-38
ECO-MSG.	4-41
A Program That Uses Full Error Checking	4-45

P

roper database management requires that all logical sequences of statements that modify the database continue successfully until completion. For example, when you update a customer account to show a reduction of \$100 in the payable balance and, for some reason, the next step (to update the cash balance) fails, your books are out of balance. Make sure you check that every SQL statement executes correctly.

This chapter discusses how to use the following diagnostic and error-handling structures:

- GET DIAGNOSTICS statement

You can use the GET DIAGNOSTICS statement to diagnose ANSI ISO and X/Open standard run-time errors in your ESQL/COBOL program. The [Informix Guide to SQL: Syntax](#) describes the structure, contents, and syntax of GET DIAGNOSTICS.

- SQLSTATE variable

GET DIAGNOSTICS uses the SQLSTATE variable to check for errors. Refer to [“The GET DIAGNOSTICS Statement” on page 4-4](#) for a description of the SQLSTATE variable.

- SQLCA record

You can use the SQLCA record to check for Informix proprietary run-time errors in your ESQL/COBOL program. Refer to [“The SQLCA Record” on page 4-17](#) for a description of the SQLCA structure.

- ECO-MSG routine

You can use the ECO-MSG routine to retrieve the message text associated with a specific Informix error number. Refer to [“ECO-MSG” on page 4-41](#) for a description of the syntax and usage of ECO-MSG.

Obtaining Diagnostic Information After an SQL Statement Executes

After you execute an SQL statement in an ESQL/COBOL program, you can obtain diagnostic information about the outcome from the GET DIAGNOSTICS statement and the SQLCA structure.

The GET DIAGNOSTICS Statement

You use the GET DIAGNOSTICS statement to diagnose error messages. After you execute an SQL statement, the database server returns a status code about that statement and stores that code in a variable called SQLSTATE. The SQLSTATE status code describes the result (or primary exception code) of the most recently executed SQL statement. You use a GET DIAGNOSTICS statement to retrieve specific diagnostic information about the SQLSTATE status code from the diagnostics area.



***Tip:** If you execute a SQL statement that generates an Informix SQLSTATE IX000 reserved error message value, you can use the GET DIAGNOSTICS statement to diagnose that error message code. The GET DIAGNOSTICS statement provides the SQLSTATE, SQLCODE, or ISAM error message text associated with that error code.*

The GET DIAGNOSTICS statement returns statement and exception information

Statement Information

The statement information GET DIAGNOSTICS returns includes the fields described in [Figure 4-1](#).

Figure 4-1
Statement-Information Fields

Field	Description
NUMBER	contains the number of exceptions that occurred on the statement.
MORE	contains the character Y for yes or N for no. Y means that the diagnostic area detected more exceptions than it stored. N means that the diagnostic area stored all the available exception information. However, an Informix Version 6.0 or 7.1 database server always returns N.
ROW_COUNT	contains the number of rows inserted, updated, or deleted when you use an INSERT, UPDATE, or DELETE SQL statement. For any other SQL statement, the value of ROW_COUNT remains undefined.

Exception Information

The exception information that GET DIAGNOSTICS returns includes the fields described in Figure 4-2.

Figure 4-2
Exception-Information Fields

Field	Description
RETURNED_SQLSTATE	contains a character string of length 5, which holds the SQLSTATE value describing the current exception.
CLASS_ORIGIN	contains a variable-length character string with a maximum length of 254, which identifies the class portion of SQLSTATE as either defined by Informix or the International Standards Organization (ISO). When ISO defines the class, the value of CLASS_ORIGIN equals 'ISO 9075'.
SUBCLASS_ORIGIN	contains a variable-length character string with a maximum length of 254, which identifies the subclass portion of SQLSTATE as either defined by Informix or the ISO. When ISO defines the subclass, the value of SUBCLASS_ORIGIN equals 'ISO 9075'.

(1 of 2)

Field	Description
MESSAGE_TEXT	contains a variable-length character string with a maximum length of 254, which contains the Informix message text describing the error condition. MESSAGE_TEXT can exist a zero-length string. This field also contains message text for any ISAM exceptions.
MESSAGE_LENGTH	contains an exact numeric value with scale 0 that represents the length of the MESSAGE_TEXT string.
SERVER_NAME	contains a character string of maximum length 254, which holds the name of the database server of the <i>current</i> connection. You can make the current connection an explicit connection (established with the CONNECT statement) or an implicit connection (established with the DATABASE, CREATE DATABASE, or START DATABASE statements). When no current connection exists, SERVER_NAME remains blank. When your current connection specifies the default database server (the database server that the INFORMIXSERVER environment variable specifies), this field also remains blank. When an application issues a DISCONNECT ALL statement and a connection fails to disconnect, this field holds the name of the database server for the connection that failed to disconnect. In addition, SERVER_NAME matches the server name found in the sqlhosts file.
CONNECTION_NAME	contains a variable-length character string with a maximum length of 254, which holds the <i>connection_name</i> of the current connection. When no current connection exists, the CONNECTION_NAME field remains blank. When the application does not specify a <i>connection_name</i> in the CONNECT statement, or when the application makes an implicit connection (a connection not established with the CONNECT statement), the CONNECTION_NAME field remains blank. When an application issues a DISCONNECT ALL and one of the connections fails to disconnect, the CONNECTION_NAME field holds the <i>connection_name</i> of the connection that failed to disconnect.

(2 of 2)



Warning: You cannot prepare the GET DIAGNOSTICS statement at run time.

Examples Illustrating the GET DIAGNOSTICS Statement

The following program illustrates how to use a GET DIAGNOSTICS statement to retrieve statement information about an SQL statement. That example shows how to obtain the number of exceptions and whether the GET DIAGNOSTICS statement detects more exceptions than it stores.

```
1  *
2  *This program, DIAG1, shows how the
3  *GET DIAGNOSTICS statement
4  *retrieves statement information.
5  *
6  IDENTIFICATION DIVISION.
7  PROGRAM-ID.
8      DIAG1.
9  *
10 ENVIRONMENT DIVISION.
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. IFXSUN.
13 OBJECT-COMPUTER. IFXSUN.
14 *
15 DATA DIVISION.
16 WORKING-STORAGE SECTION.
17 *
18 *Declare variables.
19 *
20 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
21 77 OVERF PIC X(1).
22 77 EXCEPTION-COUNT PIC S9(9) COMP-5.
23 EXEC SQL END DECLARE SECTION END-EXEC.
24 *
25 PROCEDURE DIVISION.
26 RESIDENT SECTION 1.
27 *
28 *Begin Main routine. Execute an SQL statement.
29 *Determine exceptions with GET DIAGNOSTICS.
30 *Display the exception information.
31 *
32 MAIN.
33 EXEC SQL CONNECT TO 'stores7' END-EXEC.
34 EXEC SQL GET DIAGNOSTICS :OVERF=MORE,
35      :EXCEPTION-COUNT=NUMBER END-EXEC.
36 DISPLAY 'MORE EXCEPTIONS DETECTED?: ', OVERF.
37 DISPLAY 'NUMBER OF EXCEPTIONS IS: ', EXCEPTION-COUNT.
38 EXEC SQL DISCONNECT CURRENT END-EXEC.
39 STOP RUN.
40 *
```

The following program illustrates how to use a GET DIAGNOSTICS statement to retrieve exception information about an SQL statement. This example shows how to obtain the SQLSTATE status value, the class origin, the subclass origin, the error message, and the length of the error message.

```
1  *
2  *This program, DIAG2, shows how
3  *the GET DIAGNOSTICS statement
4  *retrieves exception information.
5  *
6  IDENTIFICATION DIVISION.
7  PROGRAM-ID.
8      DIAG2.
9  *
10 ENVIRONMENT DIVISION.
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. IFXSUN.
13 OBJECT-COMPUTER. IFXSUN.
14 *
15 DATA DIVISION.
16 WORKING-STORAGE SECTION.
17 *
18 *Declare variables.
19 *
20 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
21 77 EXCEPTION-COUNT PIC S9(9) COMP-5.
22 77 COUNT-EX PIC S9(9) VALUE 1 COMP-5.
23 77 SQLSTATE PIC X(5).
24 77 CLASS-ORIGIN PIC X(254).
25 77 SUBCLASS-ORIGIN PIC X(254).
26 77 ERROR-MESS PIC X(254).
27 77 MESS-LEN PIC S9(9) COMP-5.
28 EXEC SQL END DECLARE SECTION END-EXEC.
29 *
30 PROCEDURE DIVISION.
31 RESIDENT SECTION 1.
32 *
33 *Begin Main routine. Execute an SQL statement.
34 *Determine number of exceptions and pass
35 *control to ERR-CHK subroutine to diagnose each
36 *exception.
37 *
38 MAIN.
39 EXEC SQL CONNECT TO 'stores7' END-EXEC.
40 EXEC SQL GET DIAGNOSTICS
41      :EXCEPTION-COUNT=NUMBER END-EXEC.
42 DISPLAY 'NUMBER OF EXCEPTIONS IS: ', EXCEPTION-COUNT.
43 PERFORM ERR-CHK UNTIL COUNT-EX IS GREATER THAN
```

```

44      EXCEPTION-COUNT.
45  EXEC SQL DISCONNECT CURRENT END-EXEC.
46  STOP RUN.
47  *
48  *Subroutine to diagnose each exception generated by the
49  *execution of an SQL CONNECT TO statement. Display the
50  *diagnostic information.
51  *
52  ERR-CHK.
53  EXEC SQL GET DIAGNOSTICS EXCEPTION :COUNT-EX
54      :SQLSTATE=RETURNED_SQLSTATE,
55      :CLASS-ORIGIN=CLASS_ORIGIN,
56      :SUBCLASS-ORIGIN=SUBCLASS_ORIGIN,
57      :ERROR-MESS=MESSAGE_TEXT,
58      :MESS-LEN=MESSAGE_LENGTH
59  END-EXEC.
60  DISPLAY '*****'.
61  DISPLAY 'THE SQLSTATE VALUE IS: ', SQLSTATE.
62  DISPLAY 'THE CLASS CODE ORIGIN IS: ', CLASS-ORIGIN.
63  DISPLAY 'THE SUBCLASS CODE ORIGIN IS: ', SUBCLASS-ORIGIN.
64  DISPLAY 'THE ERROR MESSAGE IS: ', ERROR-MESS.
65  DISPLAY 'THE ERROR MESSAGE LENGTH IS: ', MESS-LEN.
66      ADD 1 TO COUNT-EX.
67  *

```

Using the SQLSTATE Variable

When an SQL statement executes, your ESQL/COBOL program automatically generates an error status code that represents *success*, *failure*, *warning*, or *no data found*. Your program stores this error status code in a variable called SQLSTATE.

Class and Subclass Codes

The SQLSTATE status code, a five-character string, contains only the following elements:

- Digits
- Capital letters

The first two characters of SQLSTATE indicate a class. The last three characters of SQLSTATE indicate a subclass. Figure 4-3 shows the structure of the SQLSTATE code using the value 08001, where 08 represents the class code and 001 represents the subclass code. The value 08001 represents the following error:

Unable to connect with database environment.

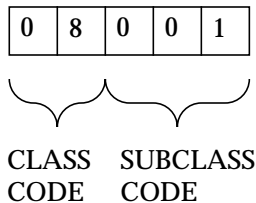


Figure 4-3
The Structure of the SQLSTATE Code

The SQLSTATE class code represents a unique category of error status conditions, but the subclass code does not. The meaning of the subclass code depends on the accompanying class code. The initial character of the class code indicates the following source of the SQLSTATE value:

- Class codes that begin with a digit in the range 0-4, or a capital letter in the range A-H, indicate that X/Open defines the result code. In this case, the associated subclass codes also begin in the range 0-4 or A-H.
- Class codes that begin with the letters IX indicate error or warning conditions only Informix uses. Informix uses codes starting with IX to support any existing warning or error messages that X/Open does not support. Other class codes that begin with a digit in the range 5-9, or a capital letter in the range I-Z, indicate currently undefined conditions.

Informix designates a class code of IX for the Informix error or warning messages that the X/Open reserved range does not support. The subclass code varies depending on the error.

List of SQLSTATE Codes

Figure 4-4 describes the class codes, subclass codes, and the meaning of all valid warning and error codes associated with the SQLSTATE error status code.

Figure 4-4
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	I00	Non-X/Open warning ('I' stands for Informix)
01	I01	Database has transactions
01	I03	ANSI-compliant database selected
01	I04	Informix OnLine database selected
01	I05	Float to decimal conversion has been used
01	I06	Informix extension to an ANSI-compliant standard syntax
01	I07	UPDATE/DELETE statement lacks a WHERE clause
01	I08	An ANSI keyword has been used as a cursor name
01	I09	The number of items in the select-list does not equal to the number in the into-list
01	I10	Database server running in secondary mode
01	I11	Dataskip is turned on
02	000	No data found

(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	Cannot execute cursor specification
07	004	Dynamic parameters require a USING clause
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 server transactions
2B	000	Dependent privilege descriptors still exist
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
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
42	000	Syntax error or access violation

(3 of 4)

Class	Subclass	Meaning
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 failure
IX	000	Informix reserved error message

(4 of 4)

Using SQLSTATE in Applications

You can use a variable, called SQLSTATE, that ESQL/COBOL does not require you to declare in your program. SQLSTATE contains the error code, essential for error handling, generated every time your program executes an SQL statement. The SQLSTATE variable exists whether or not you choose to declare it. When you do not declare SQLSTATE explicitly in your program, that program automatically and implicitly declares a variable called SQLSTATE. When you declare the SQLSTATE variable in WORKING-STORAGE, you must use the following code to retrieve the SQLSTATE status value:

```
SQL EXEC
    GET DIAGNOSTICS EXCEPTION 1
    :SQLSTATE=RETURNED_SQLSTATE
END-EXEC.
```

In this example, RETURNED_SQLSTATE contains the SQLSTATE error status code. Your ESQL/COBOL program does not need to declare an SQLSTATE variable and then store a status value in that variable using the preceding code. Your program can use an automatically declared SQLSTATE variable that already contains the status code associated with the most recently executed SQL statement. For example, when you do not declare an SQLSTATE variable, and you execute an SQL statement, you can immediately check the SQLSTATE value for that statement using the following COBOL statement:

```
DISPLAY 'THE SQLSTATE VALUE IS: ', SQLSTATE.
```

The preceding statement displays the contents of the automatically declared SQLSTATE variable even though you did not declare it.

You can examine the SQLSTATE variable to determine whether an SQL statement was successful. When the SQLSTATE variable indicates that the statement failed, you can execute a GET DIAGNOSTICS statement to obtain additional error information.

As an alternative to using the automatically generated SQLSTATE variable, you can declare a host variable within your application to receive the RETURNED_SQLSTATE value. The value in the RETURNED_SQLSTATE field of the GET DIAGNOSTICS statement provides the error code essential for error handling. You can assign this host variable any valid name you wish, including the name SQLSTATE. When you declare a host variable, however, you must explicitly issue the GET DIAGNOSTICS statement after each SQL statement that you wish to check for exceptions.

To declare an SQLSTATE variable within your application, use the following syntax:

```
WORKING-STORAGE SECTION.  
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
77 SQLSTATE PIC X(5).  
EXEC SQL END DECLARE SECTION END-EXEC.
```

For an example of how to declare and use an SQLSTATE variable in a program, refer to [“Multiple Error Conditions” on page 4-16](#).

Multiple Error Conditions

When multiple exceptions point to the same SQL statement, you can diagnose each exception. The following program shows how to use the GET DIAGNOSTICS statement to diagnose multiple exceptions:

```
1 *
2 *This program, MULTEX, executes an SQL CONNECT
3 *statement that attempts to connect to a database
4 *that does not exist. When an error occurs,
5 *the WHENEVER statement passes control
6 *to error checking subroutine.
7 *
8 IDENTIFICATION DIVISION.
9 PROGRAM-ID.
10     MULTEX.
11 *
12 ENVIRONMENT DIVISION.
13 CONFIGURATION SECTION.
14 SOURCE-COMPUTER. IFXSUN.
15 OBJECT-COMPUTER. IFXSUN.
16 *
17 DATA DIVISION.
18 WORKING-STORAGE SECTION.
19 *
20 *Declare variables.
21 *
22 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
23 77 NUM-OF-EXCEPTIONS PIC S9(9) COMP-5.
24 77 SQLSTATE-VALUE PIC X(5).
25 77 CLASS-ID PIC X(254).
26 77 SUBCLASS-ID PIC X(254).
27 77 EXPLAIN-EXCEPTION PIC X(254).
28 77 MESSAGE-TEXT-LENGTH PIC S9(3) COMP-5.
29 77 INT-VAL PIC S9(3) VALUE 1 COMP-5.
30 EXEC SQL END DECLARE SECTION END-EXEC.
31 *
32 PROCEDURE DIVISION.
33 RESIDENT SECTION 1.
34 *
35 *Begin Main routine. Main routine attempts connection
36 *to database that does not exist. SQL error occurs.
37 *Whenever statement transfers control to 100-DO-ERROR
38 *subroutine.
39 *
40 MAIN.
41 EXEC SQL WHENEVER SQLERROR GOTO 100-DO-ERROR END-EXEC.
42 DISPLAY 'THIS IS A TEST OF MULTEX'.
43 DISPLAY ' '.
44 EXEC SQL CONNECT TO 'nonexistent' END-EXEC.
45 DISPLAY 'THE SQLSTATE VALUE IS ' SQLSTATE.
46 EXEC SQL DISCONNECT ALL END-EXEC.
47 STOP RUN.
```

```

48 *
49 *Subroutine to determine and display number of exceptions generated
50 *by SQL CONNECT TO statement. 100-DO-ERROR passes control to the
51 *101-GET-DIAG subroutine.
52 *
53 100-DO-ERROR.
54 EXEC SQL GET DIAGNOSTICS :NUM-OF-EXCEPTIONS=NUMBER END-EXEC.
55 DISPLAY 'NUMBER OF EXCEPTIONS IS ' NUM-OF-EXCEPTIONS.
56 PERFORM 101-DO-GET-DIAG
57     UNTIL INT-VAL IS GREATER THAN NUM-OF-EXCEPTIONS.
58 *
59 *Subroutine to diagnose each exception. Obtain and display
60 *diagnostic information.
61 *
62 101-DO-GET-DIAG.
63 EXEC SQL GET DIAGNOSTICS EXCEPTION :INT-VAL
64     :SQLSTATE-VALUE=RETURNED_SQLSTATE,
65     :CLASS-ID=CLASS_ORIGIN,
66     :SUBCLASS-ID=SUBCLASS_ORIGIN,
67     :EXPLAIN-EXCEPTION=MESSAGE_TEXT,
68     :MESSAGE-TEXT-LENGTH=MESSAGE_LENGTH END-EXEC.
69 DISPLAY 'THE SQLSTATE VALUE IS ' SQLSTATE-VALUE.
70 DISPLAY 'THE CLASS IS ' CLASS-ID.
71 DISPLAY 'THE SUBCLASS IS ' SUBCLASS-ID.
72 DISPLAY 'THE ERROR MESSAGE IS ' EXPLAIN-EXCEPTION.
73 DISPLAY 'THE MESSAGE LENGTH IS ' MESSAGE-TEXT-LENGTH.
74 ADD 1 TO INT-VAL.
75 *

```

The SQLCA Record

After each SQL statement executes, the database server returns to the SQLCA record error status information and other information relevant to performance or to the nature of the data handled. For some statements, the database server returns warnings rather than error information. You can take advantage of this information in your ESQL/COBOL program.

ESQL/COBOL automatically includes the SQLCA record in each program. These records vary depending on the COBOL compiler.

[Figure 4-5](#) shows the SQLCA record for RM/COBOL-85 compiler.

```

77  SQLNOTFOUND PIC S9(10) VALUE 100.
01  SQLCA.
    05  SQLCODE          PIC S9(5)    COMPUTATIONAL-4.
    05  SQLERRM.
        49  SQLERRML     PIC S9(4)    COMPUTATIONAL-4.
        49  SQLERRMC     PIC X(70).
    05  SQLERRP          PIC X(8).
    05  SQLERRD          OCCURS 6 TIMES

PIC S9(5)    COMPUTATIONAL-4.
05  SQLWARN.
    10  SQLWARN0        PIC X(1).
    10  SQLWARN1        PIC X(1).
    10  SQLWARN2        PIC X(1).
    10  SQLWARN3        PIC X(1).
    10  SQLWARN4        PIC X(1).
    10  SQLWARN5        PIC X(1).
    10  SQLWARN6        PIC X(1).
    10  SQLWARN7        PIC X(1).

```

Figure 4-5
The SQLCA Record
for Ryan_McFarland
Compilers

Figure 4-6 shows the SQLCA record for MF COBOL/2 compiler.

```

77  SQLNOTFOUND PIC S9(10) VALUE 100.
01  SQLCA.
    05  SQLCODE          PIC S9(9)    COMPUTATIONAL-5.
    05  SQLERRM.
        49  SQLERRML     PIC S9(4)    COMPUTATIONAL-5.
        49  SQLERRMC     PIC X(70).
    05  SQLERRP          PIC X(8).
    05  SQLERRD          OCCURS 6 TIMES

PIC S9(9)    COMPUTATIONAL-5.
05  SQLWARN.
    10  SQLWARN0        PIC X(1).
    10  SQLWARN1        PIC X(1).
    10  SQLWARN2        PIC X(1).
    10  SQLWARN3        PIC X(1).
    10  SQLWARN4        PIC X(1).
    10  SQLWARN5        PIC X(1).
    10  SQLWARN6        PIC X(1).
    10  SQLWARN7        PIC X(1).

```

Figure 4-6
The SQLCA Record
for Micro Focus
Compilers

The following tables listed in [“The Contents of the SQLCA Structure,”](#) starting on [page 4-19](#), illustrate the principal fields of the SQLCA record.

The Contents of the SQLCA Structure

integer	SQLCODE	
0		Success.
100		No more data/not found.
negative		Error code.
array of 6 integers	SQLERRD	
SQLERRD[1]		Following successful prepare of a SELECT, UPDATE, INSERT, or DELETE statement, or after opening a select cursor, this field contains the estimated number of rows affected.
SQLERRD[2]		When SQLCODE contains an error code, this field contains either zero or an additional error code, called the ISAM error code, that explains the origin of the main error.
SQLERRD[3]		Following a successful insert operation of a single row, this field contains the value of a generated serial number for that row.
SQLERRD[4]		Following a successful, multirow insert, update, or delete operation, this field contains the count of rows processed.
SQLERRD[5]		Following a multirow insert, update, or delete operation that ends with an error, this field contains the count of rows successfully processed before the error was detected.
SQLERRD[6]		Following successful prepare of 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
character (8)	SQLERRP	
		Internal use only.

array of 8
characters

SQLWARN

SQLWARN0

SQLWARN1

SQLWARN2

SQLWARN3

SQLWARN4

SQLWARN5

SQLWARN6

SQLWARN7

When Opening a Database:

Set to *W* when a program sets any field to *W*. When **SQLWARN0** remains blank, you do not need to check the other fields.

Set to *W* when the database now open uses a transaction log.

Set to *W* when the program opens an ANSI-compliant database.

Set to *W* when using an OnLine database server.

Set to *W* when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).

Not used.

Set to *W* when the application connects to an OnLine

SQLWARN0

SQLWARN1

SQLWARN2

SQLWARN3

SQLWARN4

SQLWARN5

SQLWARN6

SQLWARN7

All Other Operations:

Set to *W* when a program sets any other field to *W*.

Set to *W* when a program truncates a column value while fetching that value into a host variable or when REVOKE ALL does not revoke all seven table-level privileges.

Set to *W* when an aggregate function encounters a null value.

On a select or on opening a cursor, set to *W* when the number of items in the select list does not equal the number of host variables given in the INTO clause to receive them. Set to *W* when GRANT ALL does not grant all seven table-level privileges.

Set to *W* when a described DELETE or UPDATE statement lacks a WHERE clause. That statement, when executed, affects all rows of the table.

character

SQLERRM

Contains the error message parameter. SQLERRM does not contain a full error message, just the parameter found within an error message. When an

Using SQLCODE OF SQLCA

Although the SQLCODE OF SQLCA value can return error values, Informix recommends that you use the SQLSTATE value for the following reasons:

- SQLSTATE complies with ANSI standards.
- You can use SQLSTATE, in association with GET DIAGNOSTICS, to return more detailed diagnostic information than SQLCODE OF SQLCA provides.

ESQL/COBOL defines the global variable SQLCODE as a long integer. Whenever the database server returns a value to SQLCODE OF SQLCA, your ESQL/COBOL program copies that value into SQLCODE. You can use SQLCODE in your INFORMIX-ESQL/COBOL program in place of SQLCODE OF SQLCA for readability and brevity.

Codes for SQL Statement Results

The database server returns the following two types of result codes after executing every SQL statement:

- SQLSTATE
- SQLCODE OF SQLCA

Figure 4-7 illustrates the outcomes and values of the preceding codes.

Figure 4-7
*SQLSTATE Values and
Related SQLCODE Values*

Outcome	SQLSTATE Class Code Value	SQLCODE OF SQLCA Value
Success	SQLSTATE = 00	SQLCODE OF SQLCA = 0
Success with warning	SQLSTATE = 01	SQLCODE OF SQLCA =SQLWARN
No data found or end of data	SQLSTATE = 02	SQLCODE OF SQLCA = SQLNOTFOUND (or 100)
Error	SQLSTATE > 02	SQLCODE OF SQLCA < 0

Success

If the SQL statement executes successfully, the database server returns an SQLSTATE class field equal to 00. The GET DIAGNOSTICS statement can use the SQLSTATE value to return other information about the SQL statement. In addition, the database server can return information to the SQLCA record. For information about the other fields in the SQLCA record, refer to [“Checking for Warnings Using the SQLWARN OF SQLCA Structure” on page 4-38](#) and the description of the SQLCA record in [“The SQLCA Record” on page 4-17](#).

Success with Warning

If the SQL statement executes successfully but generates warning conditions, the database server returns an SQLSTATE class field = 01. The GET DIAGNOSTICS statement can use the SQLSTATE value to return specific warning information. In addition, the database server can return warning information to the SQLWARN component of the SQLCA record.

No Data Found

After an SQL statement executes, you can receive a message code informing you that no data was found or the end of data has occurred. No Data Found or End of Data occurs in the following situations:

- A FETCH statement fetches no row. The implicit movement of the cursor failed because the cursor was already at the end of the set.
- A SELECT statement attempts to retrieve data from a table that has no rows.
- An unsatisfied condition (specified in an INSERT, searched DELETE, or searched UPDATE statement) exists.

SQLSTATE Class Code = 02

The SQLSTATE class field can contain a value of 02. The class field value 02 indicates that your program retrieves no more rows. The SQLSTATE No Data Found result means that the statement executed successfully but no rows satisfy the conditions of the SQL statement.

SQLCODE OF SQLCA = 100

After a fetch, SQLCODE of SQLCA can contain the value 100. The value 100 indicates that your program retrieves no more rows. The FETCH statement provides a special case with respect to SQLCA error handling. After a fetch, SQLCODE OF SQLCA can contain the values 0, 100, or a negative value. The zero and negative values indicate success and failure, respectively, as they do after the execution of other statements. The value 100 indicates that your program retrieves no more rows. For readability, ESQL/COBOL defines the value 100 as SQLNOTFOUND. When you check for SQLCODE OF SQLCA = SQLNOTFOUND, you can write code to process the results of queries only when the database server returns rows.

In an ANSI-compliant database, when any of the following statements fails to access any rows, the database server sets SQLCODE OF SQLCA equal to 100:

- INSERT INTO *table-name* SELECT ... WHERE ...
- SELECT INTO TEMP ... WHERE ...
- DELETE ... WHERE ...
- UPDATE ... WHERE ...

If a PREPARE statement contains multiple statements with a WHERE clause that does not return any rows, the database server sets SQLCODE OF SQLCA equal to 100 for the four statements shown in the preceding list. This occurs whether your program accesses ANSI or non-ANSI databases.

In the following example, the INSERT statement inserts into the **hot_items** table any **stock** item ordered in a quantity greater than 10,000. When no order exists for items in that great a quantity, the SELECT part of the statement fails to insert any rows. The database server returns SQLNOTFOUND (100) in an ANSI-compliant database, or 0 in a non ANSI-compliant database.

```
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
END-EXEC.
```

The following example of an UPDATE statement fails to update any rows when a manufacturer with the **manu_code** SWK does not exist. The database server returns SQLNOTFOUND (100) in an ANSI-compliant database, or 0 in a database that is not ANSI-compliant.

```
EXEC SQL
    UPDATE STOCK
    SET UNIT_PRICE = UNIT_PRICE * 1.05
    WHERE MANU_CODE = 'SWK'
END-EXEC.
```

Error

If the SQL statement does not execute correctly, the database server sets the SQLSTATE class field value greater than 02 and SQLCODE of SQLCA to a negative value. The database server can also set other fields in the SQLCA record, and returns an ISAM error in the diagnostic area.

Errors After a PREPARE Statement

Usually, when a PREPARE statement fails with SQLCODE < 0, a syntax error occurred in the prepared text. When a PREPARE statement fails, the SQLCA.SQLERRD[5] variable captures the offset into the text where the error occurs. Your program can use the value in SQLCA.SQLERRD[5] to indicate the approximate location of the incorrect syntax in the dynamically prepared text. ESQL/COBOL ignores spaces and tabs because they are condensed. When you use PREPARE with several statements, ESQL/COBOL returns the error status on the first error in the text, even when several errors occur.

Errors After an EXECUTE Statement

If an EXECUTE statement fails with SQLCODE < 0, a statement inside the prepared text failed. The SQLCODE variable holds the error that the database server returned from the failed statement. When SQLCODE equals 0 after the completion of an EXECUTE statement, the prepared statement (or multiple prepared statements) succeeded.

When an Error Occurs on GET DIAGNOSTICS

If an error occurs for the GET DIAGNOSTICS statement, ESQL/COBOL sets SQLCODE to the value of the exception number that generated the error. However, SQLSTATE and the values of all other fields that GET DIAGNOSTICS returns remain undefined.

Error Handling in Programs

This section describes how to check for errors and warnings in programs. You can check for errors using the GET DIAGNOSTICS statement, in-line code, and the WHENEVER statement. You can check for warnings using the GET DIAGNOSTICS statement or the SQLWARN of SQLCA structure.

Checking for Errors with the GET DIAGNOSTICS Statement

For information about checking for errors using the GET DIAGNOSTICS statement, refer to [“The GET DIAGNOSTICS Statement” on page 4-4](#).

Checking for an Error Using In-Line Code

To check for an error, test the value of `SQLSTATE` after an SQL statement executes. For example, When you want to check that a `CONNECT` statement executed as expected, use the code shown in the following program:

```
1  *
2  *This program, COBERR1, tests the value of SQLSTATE
3  *after an SQL statement executes. This program checks
4  *the execution of a CONNECT statement.
5  *
6  IDENTIFICATION DIVISION.
7  PROGRAM-ID.
8      COBERR1.
9  *
10 ENVIRONMENT DIVISION.
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. IFXSUN.
13 OBJECT-COMPUTER. IFXSUN.
14 *
15 DATA DIVISION.
16 WORKING-STORAGE SECTION.
17 *
18 PROCEDURE DIVISION.
19 RESIDENT SECTION 1.
20 *
21 *Begin Main routine. Execute an SQL statement
22 *and pass control to 100-ERROR-CHECK
23 *subroutine.
24 *
25 MAIN.
26 DISPLAY 'THIS IS A TEST OF COBERR1'.
27 DISPLAY ' '.
28 EXEC SQL
29     CONNECT TO 'nonexistent'
30 END-EXEC.
31 PERFORM 100-ERROR-CHECK.
32 EXEC SQL DISCONNECT ALL END-EXEC.
33 STOP RUN.
34 *
35 *Subroutine to check for errors. If SQLCODE does not
36 *equal ZERO (Success), display an error message.
37 *
38 100-ERROR-CHECK.
39 *
40 DISPLAY 'RUNNING ERROR CHECK ROUTINE'.
41 DISPLAY ' '.
42 IF SQLCODE IS NOT EQUAL TO ZERO
```

```

43 DISPLAY 'ERROR ' SQLSTATE
44 DISPLAY 'IN USING DATABASE'
45 DISPLAY ' '.
46 *

```

Alternatively, imagine writing a routine to process any error. Your program can call the error routine each time that SQLSTATE returned with an SQLSTATE error code not equal to 00000 (Success). The 100-ERROR-CHECK routine, part of the following program retrieves the message associated with an error. It also checks for other information about the error. It prints the error message and other diagnostic information and then exits the program.

```

1  *
2  *This program, COBERR2, retrieves diagnostic
3  *information associated with an SQL statement.
4  *
5  IDENTIFICATION DIVISION.
6  PROGRAM-ID.
7      COBERR2.
8  *
9  ENVIRONMENT DIVISION.
10 CONFIGURATION SECTION.
11 SOURCE-COMPUTER. IFXSUN.
12 OBJECT-COMPUTER. IFXSUN.
13 *
14 DATA DIVISION.
15 WORKING-STORAGE SECTION.
16 *
17 *Declare variables.
18 *
19 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
20 77 MORE-EXCEPTIONS PIC X(1).
21 77 NUM-OF-EXCEPTIONS SQLINT.
22 77 ROWS-PROCESSED PIC S9(9) COMP-5.
23 77 SQLSTATE-VALUE PIC X(5).
24 77 CLASS-ID PIC X(254).
25 77 SUBCLASS-ID PIC X(254).
26 77 EXPLAIN-EXCEPTION PIC X(254).
27 77 MESSAGE-TEXT-LENGTH PIC S9(3) COMP-5.
28 77 SERVER-VALUE PIC X(254).
29 77 NAME-OF-CONNECTION PIC X(254).
30 77 COUNT-EX PIC S9(9) VALUE 1 COMP-5.
31 EXEC SQL END DECLARE SECTION END-EXEC.
32 *
33 77 ZERO-FIELD PIC X(5) VALUE "00000".
34 *
35 PROCEDURE DIVISION.
36 RESIDENT SECTION 1.

```

Checking for an Error Using In-Line Code

```
37 *
38 *Begin Main routine. CONNECT TO generates an error
39 *because the database "nonexistent" does not exist.
40 *Pass control to 100-ERROR-CHECK subroutine to
41 *check for exceptions.
42 *
43 MAIN.
44 DISPLAY 'THIS IS A TEST OF COBOLERR2'.
45 DISPLAY ' '.
46 EXEC SQL
47     CONNECT TO 'nonexistent'
48 END-EXEC.
49 IF SQLSTATE NOT EQUAL TO ZERO-FIELD
50     PERFORM 100-ERROR-CHECK.
51 EXEC SQL DISCONNECT ALL END-EXEC.
52 STOP RUN.
53 *
54 *Subroutine to determine exceptions associated with
55 *the SQL statement in the main routine. Display
56 *exception information. Pass control to
57 *200-DIAGNOSE subroutine to diagnose each exception
58 *
59 100-ERROR-CHECK.
60 DISPLAY 'RUNNING GET DIAGNOSTICS FOR '.
61 DISPLAY 'MORE INFORMATION ON ERROR'.
62 EXEC SQL GET DIAGNOSTICS
63     :MORE-EXCEPTIONS=MORE,
64     :NUM-OF-EXCEPTIONS=NUMBER,
65     :ROWS-PROCESSED=ROW_COUNT
66 END-EXEC.
67 DISPLAY 'ARE THERE MORE EXCEPTIONS? Y/N: ',
68     MORE-EXCEPTIONS.
69 DISPLAY 'THE NUMBER OF EXCEPTIONS IS/ARE: ',
70     NUM-OF-EXCEPTIONS.
71 DISPLAY 'THE NUMBER OF ROWS PROCESSED IS: ',
72     ROWS-PROCESSED.
73 DISPLAY '*****'.
74 PERFORM 200-DIAGNOSE UNTIL COUNT-EX IS
75     GREATER THAN NUM-OF-EXCEPTIONS.
76 *
77 *Subroutine to diagnose each exception associated with
78 *the SQL statement in the main routine. Display
79 *diagnostic information.
80 *
81 200-DIAGNOSE.
82 EXEC SQL GET DIAGNOSTICS EXCEPTION :COUNT-EX
83     :SQLSTATE-VALUE=RETURNED_SQLSTATE,
84     :CLASS-ID=CLASS_ORIGIN,
85     :SUBCLASS-ID=SUBCLASS_ORIGIN,
```

```
85      :EXPLAIN-EXCEPTION=MESSAGE_TEXT,  
86      :MESSAGE-TEXT-LENGTH=MESSAGE_LENGTH,  
87      :SERVER-VALUE=SERVER_NAME,  
88      :NAME-OF-CONNECTION=CONNECTION_NAME  
89  END-EXEC.  
90  DISPLAY '*****EXCEPTION ', COUNT-EX, '*****'.  
91  DISPLAY 'THE VALUE OF SQLSTATE IS: ',  
92      SQLSTATE-VALUE.  
93  DISPLAY 'THE CLASS ORIGIN IS: ', CLASS-ID.  
94  DISPLAY 'THE SUBCLASS ORIGIN IS: ', SUBCLASS-ID.  
95  DISPLAY 'THE EXPLANATION OF THIS EXCEPTION IS: ',  
96      EXPLAIN-EXCEPTION.  
97  DISPLAY 'THE LENGTH OF THE EXPLANATION MESSAGE IS: ',  
98      MESSAGE-TEXT-LENGTH.  
99  DISPLAY 'THE VALUE OF THE SERVER IS: ', SERVER-VALUE.  
100 DISPLAY 'THE NAME OF THE CONNECTION IS: ',  
101     NAME-OF-CONNECTION.  
102 ADD 1 TO COUNT-EX.  
103 *
```

Make sure you check the status of the SQLSTATE value after each SQL statement. You can use the WHENEVER statement to reduce the amount of potential code that you write to check for errors. The following section explores how to use the WHENEVER statement.

Automatically Checking for Errors Using the WHENEVER Statement

You can use the WHENEVER statement to trap all errors and warnings that occur during the execution of SQL statements. Use the WHENEVER statement to replace the conditional test of the SQLCODE value after each SQL statement.

Use the WHENEVER statement to check for errors, SQLNOTFOUND, or warnings. You can direct the program to take any of the following actions:

- Continue execution
- Stop execution
- Execute a call
- Go to a specified section of code
- Perform a specified paragraph

For example, when you want to go to a 100-ERROR-CHECK routine every time an error occurs in a paragraph, put the following statement in the early part of the paragraph, before any SQL statements:

```
EXEC SQL WHENEVER ERROR GOTO 100-ERROR-CHECK END-EXEC.
```

Instead of using a GOTO instruction in the WHENEVER statement, you can use the CALL keyword to call a routine, or the PERFORM keyword to perform a specific paragraph. You can also use the STOP keyword to exit from the program.

The GOTO keyword remains ANSI-compliant when used in the WHENEVER statement. ESQL/COBOL designates the CALL, PERFORM, and STOP options as Informix extensions to the ANSI standard.

The following program, WHENCHK, shows some ways you can use the WHENEVER statement in your INFORMIX-ESQL/COBOL program. The DISPLAY statements explain the actions of the WHENCHK program.

```
1      *This program, WHENCHK, shows how
2      *to use the ESQL/COBOL WHENEVER statement
3      *
4      IDENTIFICATION DIVISION.
5      PROGRAM-ID.
6          WHENCHK.
7      *
8      ENVIRONMENT DIVISION.
9      CONFIGURATION SECTION.
10     SOURCE-COMPUTER. IFXSUN.
11     OBJECT-COMPUTER. IFXSUN.
12     *
13     DATA DIVISION.
14     WORKING-STORAGE SECTION.
15     *
16     *Declare variables.
17     *
18     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
19     77 RETVAL    PIC X(10).
20     EXEC SQL END DECLARE SECTION END-EXEC.
21     *
22     PROCEDURE DIVISION.
23     RESIDENT SECTION 1.
24     *
25     MAIN.
26
27         EXEC SQL
28             WHENEVER SQLERROR PERFORM SQLERR-CHECK
29             END-EXEC.
30
31         DISPLAY '*****'.
32         DISPLAY 'Connecting to default database server.'.
33
34         EXEC SQL CONNECT TO DEFAULT END-EXEC.
35
36         DISPLAY '*****'.
37         DISPLAY 'Testing WHENEVER SQLERROR.'.
38         DISPLAY 'Opening a database that does not exist.'.
39
40         EXEC SQL DATABASE 'nonexistent' END-EXEC.
41
42         DISPLAY '*****'.
43         DISPLAY 'Executing WHENEVER SQLERROR CONTINUE.'.
44
45         EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
46
47         DISPLAY '*****'.
48         DISPLAY 'Executing WHENEVER NOT FOUND.'.
49
50     EXEC SQL
```

Automatically Checking for Errors Using the *WHENEVER* Statement

```
51          WHENEVER NOT FOUND PERFORM NO-DATA-PRINTOUT
52      END-EXEC.
53
54      DISPLAY '*****'.
55      DISPLAY 'Creating an empty database called nodata.'.
56
57      EXEC SQL
58          CREATE DATABASE nodata
59      END-EXEC.
60
61      DISPLAY '*****'.
62      DISPLAY 'Creating empty table in database.'.
63
64      EXEC SQL
65          CREATE TABLE empty (emptycol CHAR)
66      END-EXEC.
67
68      DISPLAY '*****'.
69      DISPLAY 'Testing WHENEVER NOT FOUND.'.
70      DISPLAY 'Selecting table containing no data.'.
71
72      EXEC SQL
73          SELECT * FROM empty
74      END-EXEC.
75
76      DISPLAY '*****'.
77      DISPLAY 'nodata database not needed...'.
78      DISPLAY 'for further testing. Dropped.'.
79
80      EXEC SQL DROP DATABASE nodata END-EXEC.
81
82      DISPLAY '*****'.
83      DISPLAY 'Executing WHENEVER SQLWARNING.'.
84
85      EXEC SQL
86          WHENEVER SQLWARNING PERFORM WARN-CHECK
87      END-EXEC.
88
89      DISPLAY '*****'.
90      DISPLAY 'Creating database testwarn...'.
91      DISPLAY 'to test WHENEVER SQLWARNING.'.
92
93      EXEC SQL
94          CREATE DATABASE testwarn
95      END-EXEC.
96
97      DISPLAY '*****'.
98      DISPLAY 'Creating table and columns...'.
99      DISPLAY 'within testwarn database.'.
100
101      EXEC SQL
102          CREATE TABLE WARNTAB (COL1 CHAR(22))
103      END-EXEC.
104
```

Automatically Checking for Errors Using the WHENEVER Statement

```
105      DISPLAY '*****'.
106      DISPLAY 'Inserting a string into warntab column.'.
107
108      EXEC SQL
109          INSERT INTO WARTAB VALUES("THIS HAS TWENTY CHARS")
110      END-EXEC.
111
112      DISPLAY '*****'.
113      DISPLAY 'Testing WHENEVER SQLWARNING.'.
114      DISPLAY 'Truncating string value using SELECT.'.
115
116      EXEC SQL
117          SELECT * INTO :RETVAL FROM WARTAB
118      END-EXEC.
119
120      DISPLAY '*****'.
121      DISPLAY 'testwarn database not needed...'.
122      DISPLAY 'for further testing. Dropped.'.
123
124      EXEC SQL DROP DATABASE testwarn END-EXEC.
125
126      DISPLAY '*****'.
127      DISPLAY 'Executing WHENEVER ERROR.'.
128
129      EXEC SQL
130          WHENEVER ERROR PERFORM ERROR-DISPLAY
131      END-EXEC.
132
133      DISPLAY '*****'.
134      DISPLAY 'Testing WHENEVER ERROR.'.
135      DISPLAY 'Opening a database that does not exist.'.
136
137      EXEC SQL DATABASE 'nonexistent' END-EXEC.
138
139      DISPLAY '*****'.
140      DISPLAY 'Calling perform statement to show...'.
141      DISPLAY 'scope limitation of WHENEVER statement.'.
142
143      PERFORM SQLERROR-SCOPE-TEST.
144
145      DISPLAY '*****'.
146      DISPLAY 'Program now out of scope test procedure.'.
147
148      DISPLAY 'Testing WHENEVER ERROR.'.
149      DISPLAY 'Making sure WHENEVER SQLERROR is...'.
150      DISPLAY 'out of scope. Following action is ...'.
151      DISPLAY 'intended to invoke only...'.
152      DISPLAY 'WHENEVER ERROR.'.
153      DISPLAY '*****'.
154      DISPLAY 'Opening a database that does not exist.'.
155
156      EXEC SQL DATABASE 'nonexistent' END-EXEC.
157
158      DISPLAY '*****'.
```

Automatically Checking for Errors Using the *WHENEVER* Statement

```
159         DISPLAY 'Now executing WHENEVER SQLERROR STOP.'.
160
161     EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
162
163     DISPLAY '*****'.
164     DISPLAY 'Testing WHENEVER SQLERROR STOP.'.
165     DISPLAY 'Connecting to database that does not exist.'.
166     DISPLAY 'Intended to make program terminate early.'.
167     DISPLAY '*****'.
168     DISPLAY 'Opening a database that does not exist.'.
169
170     EXEC SQL DATABASE 'nonexistent' END-EXEC.
171
172     DISPLAY 'Program over.'.
173
174     STOP RUN.
175
176 *
177     SQLERR-CHECK.
178     DISPLAY '*****'.
179     DISPLAY 'WHENEVER SQLERROR has been called.'.
180     DISPLAY 'An SQL error occurred!'.
181
182 *
183     NO-DATA-PRINTOUT.
184     DISPLAY '*****'.
185     DISPLAY 'WHENEVER NOT FOUND has been called.'.
186     DISPLAY 'No Data in the database!'.
187
188 *
189     WARN-CHECK.
190     DISPLAY '*****'.
191     DISPLAY 'WHENEVER SQLWARNING has been called.'.
192     DISPLAY 'The warning statement is: ', SQLWARNO OF SQLWARN.
193
194 *
195     ERROR-DISPLAY.
196     DISPLAY '*****'.
197     DISPLAY 'WHENEVER ERROR has been called.'.
198
199 *
200     SQLERROR-SCOPE-TEST.
201     DISPLAY '*****'.
202     DISPLAY 'Showing scope of WHENEVER SQLERROR.'.
203     DISPLAY 'This statement will control error events...'.
204     DISPLAY 'in this procedure only, and return control...'.
205     DISPLAY 'to the preceding WHENEVER statement...'.
206     DISPLAY 'in the main procedure.'.
207     DISPLAY '*****'.
208     DISPLAY 'Executing WHENEVER SQLERROR.'.
209     EXEC SQL
210         WHENEVER SQLERROR PERFORM SQLERR-CHECK
211     END-EXEC.
212     DISPLAY '*****'.
213     DISPLAY 'Testing WHENEVER SQLERROR scope.'.
214     DISPLAY 'Connecting to database that does not exist.'.
215     EXEC SQL DATABASE 'nonexistent' END-EXEC.
```

For details of the syntax and use of the WHENEVER statement, refer to the [Informix Guide to SQL: Syntax](#).

Checking for Warnings Using GET DIAGNOSTICS

You can check for warnings with the SQLSTATE value and the GET DIAGNOSTICS statement after the execution of each SQL statement. The GET DIAGNOSTICS section in the [Informix Guide to SQL: Syntax](#) discusses the contents of the SQLSTATE warning value.

The database server generates the following SQLSTATE warning values for an SQL statement that produces a warning. The following list provides describes the meaning of each value:

01000	Success with warning
01002	Disconnect error; transaction rolled back
01003	Null value eliminated in set function
01004	String data, right truncation
01005	Insufficient item descriptor areas
01006	Privilege not revoked

The following program, WARN, executes an SQL statement that generates an SQLSTATE warning value. That program inserts a string value into a table, and truncates that value during a select from the table into a host variable. After truncation, WARN displays diagnostic information about the warning value.

```
1 *
2 *This program generates a warning and then diagnoses the warning
3 *using the GET DIAGNOSTICS statement.
4 *
5 IDENTIFICATION DIVISION.
6 PROGRAM-ID.
7     WARN.
8 *
9 ENVIRONMENT DIVISION.
10 CONFIGURATION SECTION.
11 SOURCE-COMPUTER. IFXSUN.
12 OBJECT-COMPUTER. IFXSUN.
13 *
14 DATA DIVISION.
15 WORKING-STORAGE SECTION.
16 *
17 *Declare variables.
18 *
19 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
20 77 NUM-OF-EXCEPTIONS PIC S9(9) COMP-5.
21 77 SQLSTATE-VALUE PIC X(5).
22 77 CLASS-ID PIC X(254).
23 77 SUBCLASS-ID PIC X(254).
24 77 EXPLAIN-EXCEPTION PIC X(254).
25 77 MESSAGE-TEXT-LENGTH PIC S9(3) COMP-5.
26 77 EX-NUM PIC S9(9) COMP-5.
27 77 COUNTER PIC S9(9) VALUE 1 COMP-5.
28 77 RETVAL PIC X(10).
29 EXEC SQL END DECLARE SECTION END-EXEC.
30 *
31 PROCEDURE DIVISION.
32 RESIDENT SECTION 1.
33 *
34 *Begin Main routine. This program generates a warning
35 *message about data truncation as follows:
36 *Establish a connection to the
37 *default database server. Create a database. Create
38 *a table. Insert a value into the databale. SELECT the
39 *table balue into a host variable that is too small to
40 *contain the value. Perform error checking on the
41 *SELECT by passing control to the ERR-CHK subroutine.
42 *Obtain exception and statement information on the
43 *warning condition. CLOSE and DROP the database.
44 *Terminate the connection.
45 *
46 MAIN.
47 EXEC SQL CONNECT TO DEFAULT END-EXEC.
```

```
48 EXEC SQL CREATE DATABASE warnmsgs END-EXEC.
49 EXEC SQL
50     CREATE TABLE WARTAB (COL1 CHAR(22))
51 END-EXEC.
52 EXEC SQL
53     INSERT INTO WARTAB VALUES("This has twenty chars")
54 END-EXEC.
55 EXEC SQL
56     SELECT * INTO :RETVAL FROM WARTAB
57 END-EXEC.
58 PERFORM EX-CHK.
59 EXEC SQL CLOSE DATABASE END-EXEC.
60 EXEC SQL DROP DATABASE warnmsgs END-EXEC.
61 EXEC SQL DISCONNECT ALL END-EXEC.
62 STOP RUN.
63 *
64 *Subroutine to determine and display the number of exceptions.
65 *EX-CHK passes control to ERR-CHK.
66 *
67 EX-CHK.
68 EXEC SQL GET DIAGNOSTICS :NUM OF EXCEPTIONS=NUMBER
69 END-EXEC.
70 DISPLAY 'THE NUMBER OF EXCEPTIONS IS: ', NUM OF EXCEPTIONS.
71 DISPLAY ' '.
72 DISPLAY '*****'.
73 PERFORM ERR-CHK UNTIL COUNTER IS GREATER THAN
74     NUM-OF-EXCEPTIONS.
75 *
76 *Obtain and display diagnostic information about each
77 *warning exception.
78 *
79 ERR-CHK.
80 EXEC SQL GET DIAGNOSTICS EXCEPTION :COUNTER
81     :SQLSTATE-VALUE=RETURNED_SQLSTATE,
82     :CLASS-ID=CLASS_ORIGIN,
83     :SUBCLASS-ID=SUBCLASS_ORIGIN,
84     :EXPLAIN-EXCEPTION=MESSAGE_TEXT,
85     :MESSAGE-TEXT-LENGTH=MESSAGE_LENGTH END-EXEC.
86 DISPLAY '*****'.
87 DISPLAY 'THE SQLSTATE VALUE IS ', SQLSTATE-VALUE.
88 DISPLAY 'THE CLASS IS COMPLIANT TO ', CLASS-ID.
89 DISPLAY 'THE SUBCLASS IS COMPLIANT TO ', SUBCLASS-ID.
90 DISPLAY 'THE ERROR MESSAGE IS ', EXPLAIN-EXCEPTION.
91 DISPLAY 'THE MESSAGE LENGTH IS ', MESSAGE-TEXT-LENGTH.
92 ADD 1 TO COUNTER.
93 *
```

Checking for Warnings Using the SQLWARN OF SQLCA Structure

The database server sets the SQLWARN0 field of the SQLWARN OF SQLCA structure to W for all warning conditions. In addition, the database server sets a second field in the SQLWARN OF SQLCA structure depending on the specific warning. Refer to [“The SQLCA Record” on page 4-17](#) for more information on the SQLWARN fields.

The following example program, COBERR3, shows how to test for warnings using the SQLWARN OF SQLCA structure. The COBERR3 program is closely related to the WARN program listed in [“Checking for Warnings Using GET DIAGNOSTICS” on page 4-35](#). The COBERR3 program differs from the WARN program because COBERR3 uses the SQLWARN OF SQLCA structure instead of the GET DIAGNOSTICS statement to handle and diagnose warnings. To test for warnings using in-line code, check to make sure your ESQL/COBOL program set the first warning field (SQLWARN0) to W. You also can use the WHENEVER statement with the SQLWARNING keyword to test when the database server issued any warnings. Once you know that the database server issued a warning, check the values of the fields in the SQLWARN group item to determine the exact nature of the warning.

In the following program, when you truncate data to fit into a CHARACTER host variable, your ESQL/COBOL program sets the SQLWARN1 and SQLWARN0 warning flags, creates a database and a table, and generates warnings on the truncation of the string value.

```
1  *
2  *This program, COBERR3, truncates data.
3  *The truncation generates a warning.
4  *The warning information is provided by the
5  *SQLCA structure.
6  *
7  IDENTIFICATION DIVISION.
8  PROGRAM-ID.
9      COBERR3.
10 *
11 ENVIRONMENT DIVISION.
12 CONFIGURATION SECTION.
13 SOURCE-COMPUTER. IFXSUN.
14 OBJECT-COMPUTER. IFXSUN.
15 *
16 DATA DIVISION.
17 WORKING-STORAGE SECTION.
18 *
19 *Declare variables.
20 *
21 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
22 77 RETVAL          PIC X(10).
23 77 SLEN            PIC S9(9) USAGE COMP-5.
24 EXEC SQL END DECLARE SECTION END-EXEC.
25 *
26 PROCEDURE DIVISION.
27 RESIDENT SECTION 1.
28 *
29 *Begin Main routine to create a string value
30 *and put it into variable too small to hold
31 *all of it. Pass control to 200-D0-WARN subroutine to
32 *diagnose warning message.
33 *
34 MAIN.
35 DISPLAY 'THIS IS A TEST OF COBERR3'.
36 DISPLAY ' '.
37 EXEC SQL CONNECT TO DEFAULT END-EXEC.
38 EXEC SQL
39     CREATE DATABASE WARNMSG
40 END-EXEC.
41 *
42 EXEC SQL
43     CREATE TABLE WARNTAB (COL1 CHAR(22))
```

Checking for Warnings Using the SQLWARN OF SQLCA Structure

```
44  END-EXEC.
45  *
46  EXEC SQL
47      INSERT INTO WARNTAB VALUES("THIS HAS TWENTY CHARS")
48  END-EXEC.
49  *
50  EXEC SQL
51      SELECT * INTO :RETVAL FROM WARNTAB
52  END-EXEC.
53  *
54  MOVE 70 TO SLEN.
55  *
56  IF SQLWARN1 OF SQLWARN OF SQLCA = 'W'
57      PERFORM 200-DO-WARN.
58  *
59  EXEC SQL
60      DROP DATABASE WARNMSG
61  END-EXEC.
62  *
63  EXEC SQL DISCONNECT ALL END-EXEC.
64  STOP RUN.
65  *
66  *Subroutine to check warning messages regarding
67  *truncation of data.
68  *
69  200-DO-WARN.
70  CALL ECO-MSG USING SQLCA, SQLERRM, SLEN, SQLCODE.
71  DISPLAY 'TRUNCATION OCCURRED'.
72  DISPLAY 'THE DATA IN THE DATABASE IS...'.
73  DISPLAY 'THIS HAS TWENTY CHARS.'
74  DISPLAY 'THE TRUNCATED VALUE IS ', RETVAL.
75  DISPLAY 'SQLWARNO OF SQLWARN IS: ', SQLWARNO OF SQLWARN.
76  DISPLAY 'SQLWARN1 OF SQLWARN IS: ', SQLWARN1 OF SQLWARN.
77  *
```

ECO-MSG

Purpose

Use the ECO-MSG message lookup routine to convert an Informix error message number into the corresponding message text string.

Syntax

```
CALL ECO-MSG USING CA, S, S-LEN, STATUS.
```

<i>CA</i>	the SQLCA record
<i>S</i>	the error message character string; the ECO-MSG routine merges the error message with any error message parameters and stores the result into <i>S</i>
<i>S-LEN</i>	the length of <i>S</i>
<i>STATUS</i>	the error status code that ECO-MSG returns

Usage

Figure 4-8 shows the corresponding COBOL data type and COBOL description for the arguments used in the ECO-MSG routine.

Figure 4-8
*Corresponding COBOL Data
Types for ECO-MSG Variables*

Argument	COBOL Type	COBOL Description
<i>S</i>	CHARACTER	PIC X(LENGTH)
<i>S-LEN</i>	INTEGER	PIC S9(?) COMP
<i>STATUS</i>	INTEGER	PIC S9(?) COMP

In Figure 4-8, LENGTH is the length of the string in digits. The ECO-MSG routine implements the PIC S9(?) shown in the COBOL Description column as PIC S9(9) for a 4-byte integer, or as PIC S9(4) for a 2-byte integer. Refer to your COBOL compiler documentation for information on whether to implement 2- or 4-byte integers.

Figure 4-8 also uses the word COMP in the **COBOL Description** column. Figure 4-9 allows you to interpret that word and shows the COMP equivalents for the types of COBOL compilers supported in ESQ/COBOL Version 7.2.

Figure 4-9
COMP Equivalents for COBOL Compilers

COMP Equivalent	Type of COBOL Compiler
COMP-2	MF COBOL/2
COMP-5	MF COBOL/2
COMP-1	RM/COBOL-85

The ECO-MSG routine typically returns the error message number in SQLCODE OF SQLCA. The ECO-MSG routine uses the system directory (**\$INFORMIXDIR/msg**) to find error message text. Preceding portions in this chapter discuss the use of SQLCA for error handling in ESQ/COBOL.

Some of the codes, listed in the next section and returned in the *STATUS* parameter, equal five characters in length. You can correctly identify these codes only when you define the *STATUS* variable as S9(x), where x>=5.

Return Codes

0	Success.
-1227	Message file not found.
-1228	Message number not found in message file.
-1231	Cannot seek within message file.
-1232	Insufficient message buffer size.
-22234	Insufficient size for the buffer you provided. ECO-MSG truncated the result to fit the buffer.

-22239 You exceeded the temporary buffer size.
 -22275 INTERNAL ERROR: You exceeded the temporary buffer
 length.

Example

The following code fragment from the ECOMSG program tests the ECO-MSG routine. You do not declare the SQLERRM component in the working storage section because the SQLCA record does that for you.

```

1  *
2  *This program tests the ECO-MSG message lookup routine.
3  *
4  IDENTIFICATION DIVISION.
5  PROGRAM-ID.
6      ECOMSG.
7  *
8  ENVIRONMENT DIVISION.
9  CONFIGURATION SECTION.
10 SOURCE-COMPUTER. IFXSUN.
11 OBJECT-COMPUTER. IFXSUN.
12 *
13 DATA DIVISION.
14 WORKING-STORAGE SECTION.
15 *
16 *Declare variable.
17 *
18 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
19 77 SLEN PIC S9(9) USAGE COMP-5.
20 EXEC SQL END DECLARE SECTION END-EXEC.
21 EXEC SQL INCLUDE SQLCA END-EXEC.
22 *
23 PROCEDURE DIVISION.
24 RESIDENT SECTION 1.
25 *
26 *Begin Main routine. Connect to database. Call
27 *ECO-MSG routine and display error message.
28 *
29 MAIN.
30 EXEC SQL
31     CONNECT TO 'personnel'
32 END-EXEC.
33 IF SQLCODE IS LESS THAN 0

```

```
34  DISPLAY 'ERROR', SQLCODE.  
35  MOVE 70 TO SLEN.  
36  CALL ECO-MSG USING SQLCA, SQLERRM, SLEN, SQLCODE.  
37  DISPLAY 'THE ERROR MESSAGE IS ', SQLERRM.  
38  STOP RUN.  
39  *
```

Example Output

If the CONNECT statement fails, the output for the preceding code fragment displays the following message:

```
THE ERROR MESSAGE IS Database not found or no system permission.
```

A Program That Uses Full Error Checking

The GET DIAGNOSTICS statement retrieves only selected status information from the diagnostics area and does not change the contents of the diagnostics area or SQLSTATE. For more information on the syntax of the GET DIAGNOSTICS statement, refer to the [Informix Guide to SQL: Syntax](#). The following example shows how to fully use GET DIAGNOSTICS, with an application-declared SQLSTATE variable, in a complete program. This program declares host variables, executes an SQL statement, and then performs error checking using the GET DIAGNOSTICS statement. The FCHECK program determines the number of exceptions that the SQL statement returns. When no exceptions occur, the program terminates. When exceptions occur, FCHECK performs an error-checking routine called ERR-CHK for each exception. ERR-CHK determines exception information and passes the exception field values into host variables. Then, the ERR-CHK routine displays the contents of the host variables.

```

1 *This program, FCHECK, uses GET DIAGNOSTICS to diagnose the
2 *execution of an SQL statement.
3 *
4 IDENTIFICATION DIVISION.
5 PROGRAM-ID.
6     FCHECK.
7 *
8 ENVIRONMENT DIVISION.
9 CONFIGURATION SECTION.
10 SOURCE-COMPUTER. IFXSUN.
11 OBJECT-COMPUTER. IFXSUN.
12 *
13 DATA DIVISION.
14 WORKING-STORAGE SECTION.
15 *
16 *Declare variables.
17 * EXEC SQL BEGIN DECLARE SECTION END-EXEC.
18 77 MORE-EXCEPTIONS PIC X(1).
19 77 NUM-OF-EXCEPTIONS PIC S9(9) COMP-5.
20 77 ROWS-PROCESSED PIC S9(9) COMP-5.
21 77 SQLSTATE PIC X(5).
22 77 CLASS-ID PIC X(254).
23 77 SUBCLASS-ID PIC X(254).
24 77 EXPLAIN-EXCEPTION PIC X(254).
25 77 MESSAGE-TEXT-LENGTH PIC S9(3) COMP-5.
26 77 SERVER-VALUE PIC X(254).
27 77 NAME-OF-CONNECTION PIC X(254).
28 77 ERR-CNT PIC S9(9) VALUE 1 COMP-5.
29 EXEC SQL END DECLARE SECTION END-EXEC.
30 *
31 PROCEDURE DIVISION.
```

A Program That Uses Full Error Checking

```
32 RESIDENT SECTION 1.
33 *
34 *Begin Main routine. Execute an SQL statement.
35 *Obtain and display exception information.
36 *Perform error checking with the ERR-CHK
37 *subroutine.
38 *
39 MAIN.
40 EXEC SQL CONNECT TO 'stores7' END-EXEC.
41 EXEC SQL GET DIAGNOSTICS
42     :MORE-EXCEPTIONS=MORE,
43     :NUM-OF-EXCEPTIONS=NUMBER,
44     :ROWS-PROCESSED=ROW_COUNT END-EXEC.
45 DISPLAY 'ARE THERE MORE EXCEPTIONS?: ', MORE-EXCEPTIONS.
46 DISPLAY 'THE NUMBER OF EXCEPTIONS IS: ', NUM-OF-EXCEPTIONS.
47 DISPLAY 'THE NUMBER OF ROWS PROCESSED IS: ', ROWS-PROCESSED.
48 DISPLAY ' '.
49 DISPLAY '*****'.
50 DISPLAY ' '.
51 PERFORM ERR-CHK UNTIL ERR-CNT IS GREATER THAN
52     NUM-OF-EXCEPTIONS.
53 EXEC SQL DISCONNECT ALL END-EXEC.
54 DISPLAY ' '.
55 DISPLAY '-----END PROGRAM-----'.
56 STOP RUN.
57 *
58 *Subroutine to diagnose each exception caused by the
59 *execution of an SQL statement. Display diagnostic
60 *information.
61 *
62 ERR-CHK.
63 DISPLAY ' '.
64 DISPLAY '-----EXCEPTION-----'.
65 DISPLAY ' '.
66 EXEC SQL GET DIAGNOSTICS EXCEPTION :ERR-CNT
67     :SQLSTATE=RETURNED_SQLSTATE,
68     :CLASS-ID=CLASS_ORIGIN,
69     :SUBCLASS-ID=SUBCLASS_ORIGIN,
70     :EXPLAIN-EXCEPTION=MESSAGE_TEXT,
71     :MESSAGE-TEXT-LENGTH=MESSAGE_LENGTH,
72     :SERVER-VALUE=SERVER_NAME,
73     :NAME-OF-CONNECTION=CONNECTION_NAME END-EXEC.
74 DISPLAY 'EXCEPTION NUMBER: ', ERR-CNT.
75 DISPLAY 'THE VALUE OF SQLSTATE IS: ', SQLSTATE.
76 DISPLAY 'THE CLASS ORIGIN IS: ', CLASS-ID.
77 DISPLAY 'THE SUBCLASS ORIGIN IS: ', SUBCLASS-ID.
78 DISPLAY 'THE EXPLANATION OF THIS EXCEPTION IS: ',
79     EXPLAIN-EXCEPTION.
80 DISPLAY 'THE LENGTH OF THE EXPLANATION MESSAGE IS: ',
81     MESSAGE-TEXT-LENGTH.
82 DISPLAY 'THE VALUE OF THE SERVER IS: ', SERVER-VALUE.
83 DISPLAY 'THE NAME OF THE CONNECTION IS: ', NAME-OF CONNECTION.
84 ADD 1 TO ERR-CNT.
85 *
```


Working with the Database Server

Understanding Database Server Connections	5-4
Client/Server Architecture of ESQL/COBOL Applications	5-4
Connecting an ESQL/COBOL Application to a Database Server	5-6
Providing Database Information	5-7
Setting Up the sqlhosts File	5-7
Using Database Server Format Conventions	5-7
Recognizing Types of Database Server Connections	5-8
Establishing Database Server Connections	5-11
Terminating Database Server Connections	5-12
Using Callback Procedures	5-13
Routines That Work with the Database Server	5-27
ECO-SIG	5-28
ECO-SQB	5-31
ECO-SQBCB	5-32
ECO-SQD	5-35
ECO-SQE	5-37
ECO-SQS	5-41

T

his chapter contains information about database server connections, callback procedures, and INFORMIX-ESQL/COBOL routines you can use to control database server processing. The following list presents that information in order of appearance in this chapter:

- [“Understanding Database Server Connections”](#) provides information about client-server architecture and database server connections.
- [“Using Callback Procedures”](#) provides information about a feature called a callback procedure that you can use to interrupt SQL processing.
- [“Routines That Work with the Database Server”](#) provides information on the following ESQL/COBOL database server manipulation routines:
 - ❑ ECO-SIG
 - ❑ ECO-SQB
 - ❑ ECO-SQBCB
 - ❑ ECO-SQD
 - ❑ ECO-SQE
 - ❑ ECO-SQS

Understanding Database Server Connections

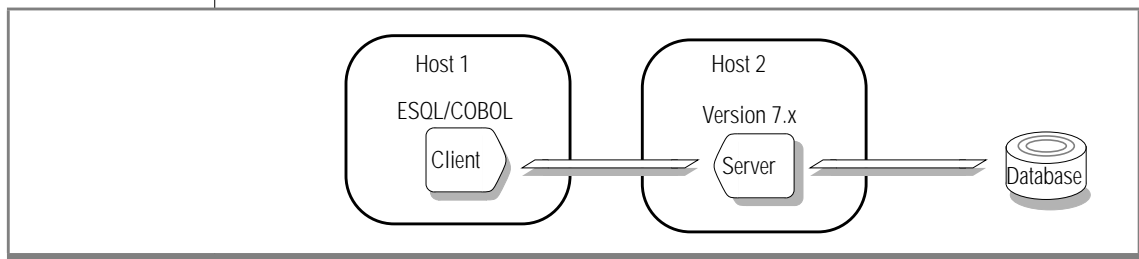
This section provides background information on methods and syntax that your INFORMIX-ESQL/COBOL program can use to establish connections to database servers. This section discusses the following information:

- “Client/Server Architecture of ESQL/COBOL Applications” discusses general client server architecture and connection concepts associated with INFORMIX-ESQL/COBOL.
- “Connecting an ESQL/COBOL Application to a Database Server” discusses the following information associated with INFORMIX-ESQL/COBOL database server connections:
 - “Providing Database Information”
 - “Setting Up the sqlhosts File”
 - “Using Database Server Format Conventions”
 - “Recognizing Types of Database Server Connections”
 - “Establishing Database Server Connections”
 - “Terminating Database Server Connections”

Client/Server Architecture of ESQL/COBOL Applications

When an INFORMIX-ESQL/COBOL program executes an SQL statement, it effectively passes the statement to a database server and receives database and status information in return. Figure 5-1 shows that client/server process.

Figure 5-1
ESQL/COBOL Client/server Process



The ESQL/COBOL program and the database server communicate with each other through an interprocess-communication mechanism. The ESQL/COBOL program represents the *client* process in the dialog because it requests information from the database server. The database server represents the *server* process because it provides information in response to requests from the client. The division of labor between the client and server processes gives you an advantage in networks where data does not reside in the same location as the client program that needs it. Figure 5-2 illustrates the possible connections between an ESQL/COBOL program and local database servers residing on the same computer.

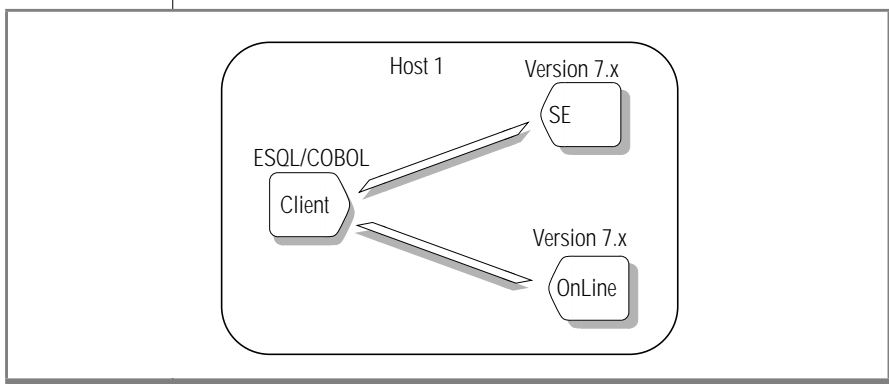


Figure 5-2
ESQL/COBOL
Connecting to an
Informix Version 7.x
Local SE Database
Server or an
Informix Version 7.x
Local OnLine
Database Server

Figure 5-3 illustrates an INFORMIX-ESQL/COBOL program connecting across a network to an Informix Version 7.x remote database server.

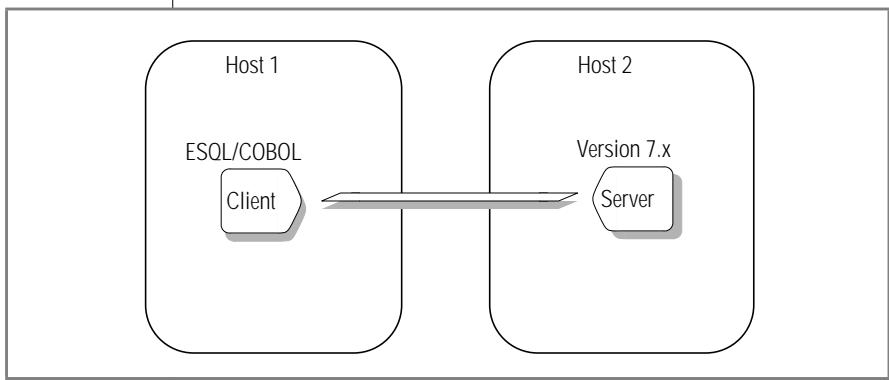


Figure 5-3
ESQL/COBOL
Connecting to an
Informix Version 7.x
Remote Database
Server

Figure 5-4 illustrates an INFORMIX-ESQL/COBOL program using the Relay Module component of an Informix Version 5.x local database server to connect across a network to an Informix Version 7.x remote database server.

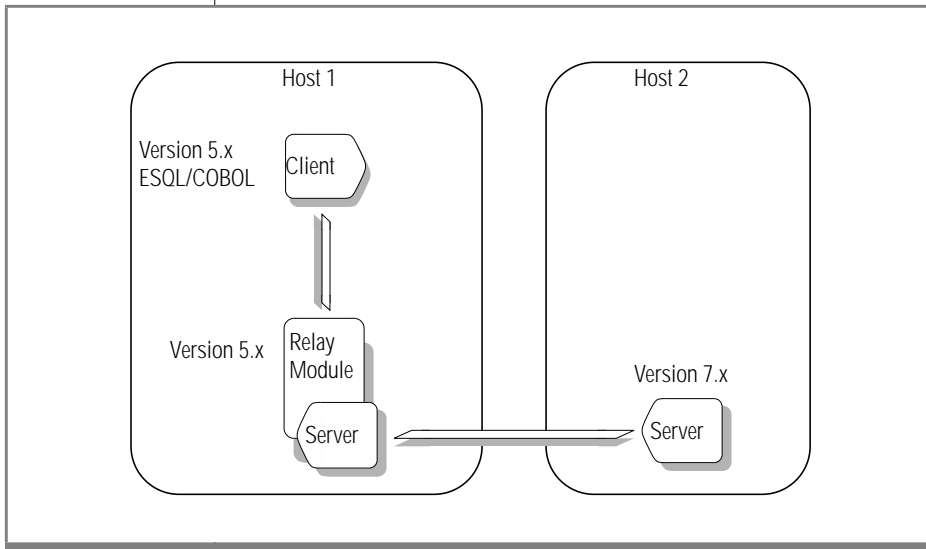


Figure 5-4
ESQL/COBOL Using the Relay Module of an Informix Version 5.x Local Database Server to Connect to an Informix Version 7.x Remote Database Server

Connecting an ESQL/COBOL Application to a Database Server

An ESQL/COBOL program includes the ability to communicate with database servers that reside either on the same computer (*local*), or over a network on other computers (*remote*). An INFORMIX-ESQL/COBOL program can connect locally or across a network to the following types of database servers:

- A default database server is the one that your INFORMIX-ESQL/COBOL program connects to automatically. To specify a default database server, set the **INFORMIXSERVER** or **DBPATH** environment variable. For more information, refer to [“Understanding a Default Connection” on page 5-10](#).
- A specific database server is the one that you specify in an INFORMIX-ESQL/COBOL statement.

Providing Database Information

Before your ESQL/COBOL application can communicate with a database server, you must provide the following database server communications information:

- The locations of the database servers on the network
- The type of interprocess-communication mechanism that the two processes use
- The name of a default database server used when the SQL statement does not name a specific database server

You describe the database servers and the corresponding interprocess-communication mechanisms in the `$INFORMIXDIR/etc/sqlhosts` file. You provide the name of the default database server in the `INFORMIXSERVER` environment variable.

Setting Up the sqlhosts File

Both the [*INFORMIX-OnLine Dynamic Server Administrator's Guide*](#) and the [*INFORMIX-SE Administrator's Guide*](#) describe the procedure to create an entry for a database server in the `sqlhosts` file. See your database administrator (DBA) to create the necessary entries in this file, particularly when no database server exists on the computer where the client program runs. In this case, you must provide an `sqlhosts` file on the host computers of both the ESQL/COBOL client program and the database server.

Using Database Server Format Conventions

You can specify a database server using several types of formats. For more information on database server naming conventions, refer to the Database Name segment in the [*Informix Guide to SQL: Syntax*](#).

Recognizing Types of Database Server Connections

Your INFORMIX-ESQL/COBOL program can establish the following types of connections:

- Multiple
- Explicit
- Implicit
- Dormant
- Current
- Default
- Specific

The following subsections describe the connections in the preceding list.

Understanding Multiple Connections

Your INFORMIX-ESQL/COBOL application can establish multiple database server connections. However, that application can communicate with only one database server at a time. You can establish multiple *dormant* connections, but only one *current* connection. For information on *dormant* connections, refer to [“Understanding a Dormant Connection” on page 5-9](#). For information on *current* connections, refer to [“Understanding a Current Connection” on page 5-9](#).

Understanding Explicit Connections

The following statements explicitly create or terminate connections:

- CONNECT TO
- SET CONNECTION
- DISCONNECT

The preceding statements are considered explicit because they exhibit different connection behavior than similar statements that predate them such as DATABASE, CREATE DATABASE, DROP DATABASE, and so on. Refer to the [Informix Guide to SQL: Syntax](#) for specific information on the behavior of the statements in the preceding list.

Understanding Implicit Connections

Implicit connections lack any association with the CONNECT TO, SET CONNECTION, and DISCONNECT statements. Only the following SQL statements or ESQL/COBOL routines implicitly create or terminate connections:

- CLOSE DATABASE statement
- CREATE DATABASE statement
- DATABASE statement
- DROP DATABASE statement
- START DATABASE statement
- ECO-SQE routine
- ECO-SQS routine

The preceding statements are considered explicit because they produce different connection behavior than CONNECT TO, SET CONNECTION, and DISCONNECT produce. Refer to the [Informix Guide to SQL: Syntax](#) for specific information on the statements in the preceding list. Refer to “ECO-SQE” on [page 5-37](#) and “ECO-SQS” on [page 5-41](#) for information on the two INFORMIX-ESQL/COBOL database server routines that appear in the preceding list.

Understanding a Current Connection

Because your application can connect to multiple database servers but cannot communicate with more than one database server at any one time, your application communicates with a database server using a *current connection*. Although communication occurs through the current connection, all other database server connections remain dormant. Refer to the [Informix Guide to SQL: Syntax](#) for specific information on the current connections.

Understanding a Dormant Connection

When your application uses a current connection to communicate with one database server, all other database server connections remain dormant. A *dormant connection* exists, but remains unused until your application makes that connection current. Refer to the [Informix Guide to SQL: Syntax](#) for specific information on dormant connections.

Understanding a Default Connection

Your application connects to the default database server when the connection statement does not specify a database server. For example, each of the following statements opens a database but none specifies a database server. In each case, you connect to the default database server prior to opening the database.

```
CONNECT TO 'stores7';  
CONNECT TO '/usr/mustang/stores7';  
DATABASE spitfire;  
CONNECT TO DEFAULT;
```

To designate a default database server, you must first set the **INFORMIX-SERVER** environment variable.

The following list describes the two types of default connections:

- An implicit default connection occurs when you establish a default connection without using **CONNECT TO DEFAULT** or **SET CONNECTION**. The following examples illustrate an implicit default connection:

```
DATABASE stores7  
CALL ECO-SQS  
START DATABASE 'stores7'  
CREATE DATABASE 'stores7'
```

The preceding statements assume that no current connection exists, or that a current default connection exists.

- An explicit default connection occurs when you establish a default connection using **CONNECT TO** or **SET CONNECTION**. The following examples illustrate an explicit default connection:

```
CONNECT TO DEFAULT  
SET CONNECTION DEFAULT  
SET CONNECTION 'stores7'
```

The last statement in the preceding example assumes that a relationship exists between database environment and the default database server.

Refer to the [Informix Guide to SQL: Syntax](#) for more information about connecting to the default database server. Refer to “[ECO-SQS](#)” on [page 5-41](#) for more information about the ECO-SQS routine.

Understanding a Specific Connection

You establish a connection to a specific database server when you name that database server in a CONNECT statement or in one of the DATABASE statements, such as DATABASE, START DATABASE, and so on. Each of the following statements establishes a connection to a specific database server called **thunderbolt**:

```
CONNECT TO 'stores7@thunderbolt';  
CONNECT TO '@thunderbolt';  
DATABASE '//thunderbolt/stores7';
```

The following list describes the two types of specific connections:

- An implicit specific connection occurs when you establish a specific connection without using CONNECT TO or SET CONNECTION. The following examples illustrate an implicit specific connection:

```
DATABASE 'stores7@thunderbolt'  
START DATABASE 'stores7@thunderbolt'  
CREATE DATABASE 'stores7@thunderbolt'
```

- An explicit specific connection occurs when you establish a specific connection using CONNECT TO or SET CONNECTION. The following examples illustrate an explicit specific connection:

```
CONNECT TO 'stores7@thunderbolt'  
SET CONNECTION 'storest7@thunderbolt'
```

Refer to the [Informix Guide to SQL: Syntax](#) for more information about connecting to a specific database server.

Establishing Database Server Connections

The following list describes the SQL statements and INFORMIX-ESQL/COBOL routines you can use to establish a database server connection.

- CONNECT TO establishes a connection to a specified database server that you specify. When you do not specify a database server, this statement establishes a connection to the default database server.
- CONNECT TO DEFAULT establishes a connection only to the default database server.

- CREATE DATABASE establishes a connection to the database server that you specify and simultaneously creates a database. When you do not specify a database server, this statement establishes a connection to the default database server.
- DATABASE establishes a connection to the database server that you specify and simultaneously selects a database. When you do not specify a database server, this statement establishes a connection to the default database server.
- ECO-SQS routine establishes a connection only to the default database server.
- SET CONNECTION reestablishes a connection. Transforms a dormant connection into a current connection.
- START DATABASE establishes a connection to the database server that you specify and simultaneously selects a database. When you do not specify a database server, this statement establishes a connection to the default database server.

For more information on the preceding SQL statements, refer to the [Informix Guide to SQL: Syntax](#). For more information on the ECO-SQS routine, refer to “ECO-SQS” on page 5-41.

Terminating Database Server Connections

The following list describes the SQL statements and INFORMIX-ESQL/COBOL routines you can use to terminate a database server connection.

- DISCONNECT ALL terminates all connections that your application established.
- DISCONNECT CURRENT terminates only the current connection.
- DISCONNECT *dbservername* terminates a specific database server connection where *dbservername* is the name of the database server from which you are disconnecting.
- DISCONNECT DEFAULT terminates all default connections.
- DROP DATABASE terminates the specified database and thus terminates the connection to that database.
- ECO-SQE routine terminates all connections that your application established.

The DISCONNECT, DISCONNECT ALL, and DISCONNECT CURRENT statements *explicitly* terminate database server connections. The remaining statements and the ECO-SQE routine in the preceding list *implicitly* terminate database server connections. For more information on the preceding SQL statements, refer to the [Informix Guide to SQL: Syntax](#). For more information on the ECO-SQE routine, refer to “ECO-SQE” on page 5-37.

Using Callback Procedures

INFORMIX-ESQL/COBOL supports the use of callback procedures to provide you with more control over database server processing. A callback procedure is a procedure you create in a file separate from your application. The callback procedure allows you to do other tasks while the database server processes an SQL task. An ESQL/COBOL callback procedure allows you to do the following:

- Check the status of SQL processing
You can use the ECO-SQD routine inside your callback procedure to make sure that the database server has finished SQL processing. In addition, you can use the ECO-SQD return code status value as a flag to trigger other events such as cancelling SQL processing, displaying an interactive menu, calling other routines, or performing other non-SQL tasks.
- Cancel SQL processing
You can use the ECO-SQB routine inside your callback procedure to cancel SQL processing.
- Display an interactive menu during SQL processing, prompting the user to choose an action
- Do other tasks, or call other routines, during SQL processing.
However, you cannot use any statements or routines that start new SQL processing.

You use the ECO-SQBCB routine to register your callback procedure before your application can begin SQL processing. This tells your application the name of the callback procedure you and the time intervals when your application calls the callback procedure. Without a callback procedure, you cannot elegantly or interactively break SQL processing while a program runs.

[Figure 5-5](#) lists the client/server process for cancelling an SQL query.

Figure 5-5
Steps for Client-Server Cancellation of an SQL Query

Step	Client Action	Database Server Action	SQL Processing
1	Execute application	Listen	None
2	Execute SQL statement	Listen	None
3	Listen	Receive SQL statement	None
4	Listen	Parse SQL statement	None
5	Listen	Listen	Begin SQL processing
6	Listen	Listen	SQL processing continues
7	Timeout interval occurs	Listen	SQL processing continues
8	Underlying communications structure passes control to callback procedure in client application	Listen	SQL processing continues
9	Client application executes callback procedure	Listen	SQL processing continues
10	Callback procedure executes ECOSQD routine to determine whether the database server is still performing SQL processing	Listen	SQL processing continues
11	Receive ECOSQD request	Listen	SQL processing continues
12	Determines whether SQL processing is still occurring	Listen	SQL processing continues

(1 of 3)

Step	Client Action	Database Server Action	SQL Processing
13	Sends return code to ECOSQD specifying whether SQL processing is still occurring	Listen	SQL processing continues.
14	Callback procedure receives ECOSQD return code	Listen	SQL processing continues
15A	If SQL processing is finished, the callback procedure terminates and passes control to the line of code following the SQL statement that started SQL processing	Listen	None
15B	If SQL processing is still occurring, the callback procedure executes the ECOSQB routine to request the database server to break SQL processing	Listen	SQL processing continues
16	The ECOSQB routine sends a request to the database server to break SQL processing	Listen	SQL processing continues
17	Listen	Receives request to break SQL processing from the ECOSQB routine	SQL processing continues
18	Listen	Starts terminating SQL processing	Begins shutting down
19	Listen	Sends message to ECOSQB routine that the database server has terminated database server processing	

(2 of 3)

Step	Client Action	Database Server Action	SQL Processing
20	The ECOSQB routine receives status notification from the database server that SQL processing is terminating.	Listen	
21	Callback procedure passes control to the line of code following the SQL statement that started SQL processing	Listen	
22	Application executes line of code following the SQL statement that started SQL processing.	Listen	
23	Application execution resumes	Listen	None

(3 of 3)

After you register your callback procedure, you create a separate callback procedure file that your application can call. Figure 5-6 shows the differences between a callback procedure and other INFORMIX-ESQL/COBOL procedures.

Figure 5-6
*Callback Procedure Compared with
Other ESQL/COBOL Procedures*

Callback procedure	Other ESQL/COBOL procedures
Must register using the ECO-SQBCB routine	Not registered
Called automatically at a specific time interval that you specify	Called only using a PERFORM statement

(1 of 2)

Callback procedure	Other ESQL/COBOL procedures
Called during SQL processing	Not called during SQL processing
Must reside in a file separate from the calling program	Can reside in calling program or in a file separate from the calling program
Is an entire file	Is only a procedure and never more than part of an entire file

(2 of 2)

The following list outlines the rules for creating a callback procedure:

1. Write the callback procedure. It must exist in a different file than the one containing the calling program where you register the callback procedure.
2. In the WORKING-STORAGE section of the callback procedure file, make sure you declare all variables used in the callback procedure.
3. The callback procedure file name must match the file name specified using the ECO-SQBCB callback registration routine located in your calling program.
4. Make sure you declare a status variable in the LINKAGE section of the callback procedure file. That status variable allows you to check the status of SQL processing while the callback procedure executes. The following list shows the three possible values and their descriptions:
 - 0 SQL processing is complete. When this value occurs, do not attempt to break SQL processing. Unregister the callback routine and return to the calling program.
 - 1 Your callback request has begun. This intermediary step requires no action.
 - 2 SQL processing is still active. When this value occurs, you can break SQL processing using the ECO-SQB routine. After you break SQL processing, unregister the callback procedure and return to the calling program.
5. Remember, a callback procedure is *not* a procedure within a file, it is an *entire* file called from your calling program.

6. Make sure you compile your callback procedure file *before* you compile the calling program. Otherwise, the calling program does not compile successfully.

The following code fragment, from the CALLPROC procedure file, shows how to write a callback procedure. First, the code fragment calls the ECO-SQD routine to see whether database processing is still active. When database processing is active, the callback procedure terminates. No other callback procedure actions occur.

However, when the ECO-SQD routine returns a value of -439, the callback procedure takes actions based on the status code variable declared in the LINKAGE section. If SQL processing is still active, the callback procedure calls ECO-SQB and breaks SQL processing. When the callback procedure ends, it unregisters ECO-SQBCB and program control returns to the calling program. The calling program, CAN-QRY, starts running again at the line following the cancelled SQL processing statement. The comments in the following example code fragment explain what the CALLPROC callback procedure does.

```
1 *
2 *Callback procedure program callproc.
3 *This procedure is called by another
4 *ESQL/COBOL program. It checks on the
5 *status of SQL processing. If SQL
6 *processing is active, it cancels SQL
7 *processing. If SQL processing is not
8 *active, it unregisters the callback
9 *function. When finished, it passes
10 *control to the calling function.
11 *
12 IDENTIFICATION DIVISION.
13 PROGRAM-ID.    CALLPROC.
14 *
15 ENVIRONMENT DIVISION.
16 CONFIGURATION SECTION.
17 SOURCE-COMPUTER.  IFXSUN.
18 OBJECT-COMPUTER.  IFXSUN.
19 *
20 DATA DIVISION.
21 *
22 WORKING-STORAGE SECTION.
23 *
```

```

24 *Declare variables. CT-TIME holds the
25 *ECO-SQBCB time-out value (milliseconds).
26 *CB-NAME holds the name of the ECO-SQBCB
27 *procedure name. CB-NAME-LEN holds the
28 *length of the CB-NAME string. ECOSQD-STAT-CODE
29 *holds the status code returned by the
30 *ECOSQD routine. SQBCB-STAT-CODE holds
31 *the status code returned by the ECO-SQBCB routine.
32 *
33 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
34 77 CB-TIME PIC S9(9) COMP-5.
35 77 CB-NAME PIC X(12).
36 77 CB-NAME-LEN PIC S9(9) COMP-5 VALUE 12.
37 77 ECOSQD-STAT-CODE PIC S9(9) COMP-5.
38 77 SQBCB-STAT-CODE PIC S9(9) COMP-5.
39 EXEC SQL END DECLARE SECTION END-EXEC.
40
41 *The following global variable, DID-IT-BREAK,
42 *tells you if SQL processing is active
43 *while a callback request is underway.
44
45 *
46 01 DID-IT-BREAK IS GLOBAL PIC S9(9).
47
48 *The following linkage variable, CB-STAT-CODE,
49 *allows you to check the status of the
50 *callback procedure.
51
52 LINKAGE SECTION.
53 01 CB-STAT-CODE PIC S9(9) COMP-5.
54 *
55 *Begin callback procedure.
56 *
57 PROCEDURE DIVISION USING CB-STAT-CODE.
58 RESIDENT SECTION 1.
59 *
60 DISPLAY ' '.
61 DISPLAY 'Callback status code is: ', CB-STAT-CODE.
62 DISPLAY ' '.
63 DISPLAY 'Callback procedure activated.'.
64 DISPLAY ' '.
65 DISPLAY 'Attempting to determine...'.
66 DISPLAY 'if SQL processing is still active.'.
67 DISPLAY '*****'.
68 *
69 *Check if SQL processing is active or not.
70 *
71 CALL ECO-SQD USING ECOSQD-STAT-CODE.
72 *
73 DISPLAY 'ECO-SQD status code is: ', ECOSQD-STAT-CODE.
74 *
75 *If SQL processing is active (-439), execute the following
76 *conditional statement.
77 *

```

```
78 IF ECOSQD-STAT-CODE IS EQUAL TO -439
79     DISPLAY '*****'
80     DISPLAY 'SQL Processing is still active.'
81     DISPLAY '*****'
82 *
83 *Check the linkage variable to see if SQL processing is
84 *still active or not.
85 *
86 *If SQL processing is not active (0), go to the
87 *last ELSE statement, unregister the callback procedure,
88 *and return to the calling program.
89 *
90     IF CB-STAT-CODE IS EQUAL TO 0
91         DISPLAY 'SQL Processing finished.'
92         DISPLAY 'No need to break SQL processing now!'
93         DISPLAY 'No more calls to Callback procedure.'
94         DISPLAY 'Sorry!'
95         DISPLAY '*****'
96 *
97 *If SQL processing is still active, and the callback
98 *request succeeded and is underway (1), move 0 into the
99 *global flag variable DID-IT-BREAK to tell the next
100 *ELSE statement that it is okay to break SQL processing
101 *and return to the calling program.
102 *
103     ELSE
104         IF CB-STAT-CODE IS EQUAL TO 1
105             DISPLAY 'Your Callback request has begun.'
106             DISPLAY 'Please be patient.'
107             DISPLAY '*****'
108             MOVE 0 TO DID-IT-BREAK
109 *
110 *If the global flag variable DID-IT-BREAK is 0, call the
111 *ECO-SQB routine to break SQL processing. Then, unregister
112 *the callback procedure and return to the calling program.
113 *
114     ELSE
115         IF DID-IT-BREAK IS EQUAL TO 0
116             DISPLAY 'SQL Processing is still active.'
117             DISPLAY 'Calling ECO-SQB to break SQL query.'
118             DISPLAY '*****'
119             CALL ECO-SQB
120             DISPLAY 'Request to break in progress.'
121             DISPLAY '*****'
122             DISPLAY 'Unregistering Callback procedure.'
123             MOVE " " TO CB-NAME
124             MOVE -1 TO CB-TIME
125             CALL ECO-SQBCB USING CB-TIME, CB-NAME,
126                 CB-NAME-LEN, SQBCB-STAT-CODE
127             DISPLAY '*****'
128             DISPLAY 'Callback procedure unregistered.'
129             DISPLAY 'ECO-SQBCB code: ', SQBCB-STAT-CODE
130             DISPLAY '*****'
131 *
```

```
132*If DID-IT-BREAK is not 0, move 1 into DID-IT-BREAK so that
133*the program does not call ECO-SQB to break SQL processing
134*the next time through the procedure. Then, return to the
135*calling program.
136*
137         ELSE
138             MOVE 1 TO DID-IT-BREAK
139*
140*If SQL Processing was not initially found to be active
141*by the first IF statement in this procedure, unregister
142*the callback procedure and return to the calling program.
143*
144     ELSE
145         DISPLAY '*****'
146         DISPLAY 'SQL Processing not active.'
147         DISPLAY 'No need to call callback procedure again.'
148         DISPLAY '*****'
149         DISPLAY 'Unregistering Callback procedure.'
150         MOVE " " TO CB-NAME
151         MOVE -1 TO CB-TIME
152         CALL ECO-SQBCB USING CB-TIME, CB-NAME,
153             CB-NAME-LEN, SQBCB-STAT-CODE
154         DISPLAY '*****'
155         DISPLAY 'Callback procedure unregistered.'
156         DISPLAY 'ECO-SQBCB code: ', SQBCB-STAT-CODE
157         DISPLAY '*****'.
158 DISPLAY '*****'.
```

The following code fragment, from the CAN-QRY program, shows how to call a callback procedure during SQL processing. The WORKING-STORAGE section sets the ECO-SQBCB *TIME-OUT* variable to 5 and forces the CAN-QRY program to call the callback routine every 5 milliseconds. The SELECT statement found in CAN-QRY is, by design, very long so that the *TIMEOUT* value occurs during SQL processing and CAN-QRY calls the callback procedure. You can find the callback procedure, CALLPROC, listed on the preceding pages.

```
1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.    CAN-QRY.
3      *
4      ENVIRONMENT DIVISION.
5      CONFIGURATION SECTION.
6      SOURCE-COMPUTER.  IFXSUN.
7      OBJECT-COMPUTER.  IFXSUN.
8      *
9      DATA DIVISION.
10     *
11     WORKING-STORAGE SECTION.
12
13     *
14     *Declare variables.
15     *
16     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17     77  CB-STAT-CODE      PIC S9(9) COMP-5.
18     77  CB-TIME          PIC S9(9) COMP-5 VALUE 5.
19     77  CB-NAME          PIC X(12) VALUE 'CALLPROC'.
20     77  CB-NAME-LEN      PIC S9(9) COMP-5 VALUE 12.
21     77  ECOSQD-STAT-CODE PIC S9(9) COMP-5.
22     77  ECOSQB-STAT-CODE PIC S9(9) COMP-5.
23     EXEC SQL END DECLARE SECTION END-EXEC.
24     *
25
26     PROCEDURE DIVISION.
27     RESIDENT SECTION 1.
28     *
29     MAIN.
30         PERFORM CONNECT-PROC.
31         PERFORM REGISTER-PROC.
32         PERFORM LONG-SQL-QUERY.
33         PERFORM DISCONNECT-PROC.
34         PERFORM CLEANUP.
35     STOP RUN.
36
37     *
38     CONNECT-PROC.
39         DISPLAY '*****'.
40         DISPLAY 'Connecting to stores7 database.'.
41         EXEC SQL CONNECT TO 'stores7@steveo16' END-EXEC.
42         DISPLAY 'Checking for database connection errors.'.
43         PERFORM MAIN-ERROR-CHECK.
```

```

44      *
45      REGISTER-PROC.
46          DISPLAY '*****'.
47          DISPLAY 'Registering Procedure.'.
48          CALL ECO-SQBCB USING CB-TIME, CB-NAME,
49              CB-NAME-LEN, CB-STAT-CODE.
50          DISPLAY 'Checking for registration errors.'.
51          PERFORM MAIN-ERROR-CHECK.
52      *
53      DISCONNECT-PROC.
54          DISPLAY '*****'.
55          DISPLAY 'Disconnecting stores7 database.'.
56          EXEC SQL DISCONNECT 'stores7@steveo16' END-EXEC.
57          DISPLAY 'Checking for disconnect errors.'.
58          PERFORM MAIN-ERROR-CHECK.
59      *
60      CLEANUP.
61          DISPLAY '*****'.
62          DISPLAY 'Program over.'.
63      *
64      LONG-SQL-QUERY.
65          DISPLAY '*****'.
66          DISPLAY 'Performing long select.'.
67          EXEC SQL
68              SELECT * FROM stock a0, stock a1, stock a2, stock a3,
69              stock a4, stock a5, stock a6, stock a7, stock a8,
70              stock a9, stock b0, stock b1, stock b2, stock b3,
71              stock b4, stock b5, stock b6, stock b7, stock b8,
72              stock b9, stock c0, stock c1, stock c2, stock c3,
73              stock c4, stock c5, stock c6, stock c7, stock c8,
74              stock c9, stock d0, stock d1, stock d2, stock d3,
75              stock d4, stock d5, stock d6, stock d7, stock d8,
76              stock d9, stock e0, stock e1, stock e2, stock e3,
77              stock e4, stock e5, stock e6, stock e7, stock e8,
78              stock e9, stock f0, stock f1, stock f2, stock f3,
79              stock f4, stock f5, stock f6, stock f7, stock f8,
80              stock f9, stock g0, stock g1, stock g2, stock g3,
81              stock g4, stock g5, stock g6, stock g7, stock g8,
82              stock g9, stock h0, stock h1, stock h2, stock h3,
83              stock h4, stock h5, stock h6, stock h7, stock h8,
84              stock h9, stock i0, stock i1, stock i2, stock i3,
85              stock i4, stock i5, stock i6, stock i7, stock i8,
86              stock i9, stock j0, stock j1, stock j2, stock j3,
87              stock j4, stock j5, stock j6, stock j7, stock j8,
88              stock j9, stock k0, stock k1, stock k2, stock k3,
89              stock k4, stock k5, stock k6, stock k7, stock k8,
90              stock k9, stock l0, stock l1, stock l2, stock l3,
91              stock l4, stock l5, stock l6, stock l7, stock l8,
92              stock l9, stock m0, stock m1, stock m2, stock m3,
93              stock m4, stock m5, stock m6, stock m7, stock m8,
94              stock m9, stock n0, stock n1, stock n2, stock n3,
95              stock n4, stock n5, stock n6, stock n7, stock n8,
96              stock n9, stock o0, stock o1, stock o2, stock o3,
97              stock o4, stock o5, stock o6, stock o7, stock o8,

```

```

98      stock o9, stock p0, stock p1, stock p2, stock p3,
99      stock p4, stock p5, stock p6, stock p7, stock p8,
100     stock p9, stock q0, stock q1, stock q2, stock q3,
101     stock q4, stock q5, stock q6, stock q7, stock q8,
102     stock q9, stock r0, stock r1, stock r2, stock r3,
103     stock r4, stock r5, stock r6, stock r7, stock r8,
104     stock r9, stock s0, stock s1, stock s2, stock s3,
105     stock s4, stock s5, stock s6, stock s7, stock s8,
106     stock s9, stock t0, stock t1, stock t2, stock t3,
107     stock t4, stock t5, stock t6, stock t7, stock t8,
108     stock t9, stock u0, stock u1, stock u2, stock u3,
109     stock u4, stock u5, stock u6, stock u7, stock u8,
110     stock u9, stock v0, stock v1, stock v2, stock v3,
111     stock v4, stock v5, stock v6, stock v7, stock v8,
112     stock v9, stock w0, stock w1, stock w2, stock w3,
113     stock w4, stock w5, stock w6, stock w7, stock w8,
114     stock w9, stock x0, stock x1, stock x2, stock x3,
115     stock x4, stock x5, stock x6, stock x7, stock x8,
116     stock x9, stock z0, stock z1, stock z2, stock z3,
117     stock z4, stock z5, stock z6, stock z7, stock z8,
118     stock z9
119     END-EXEC.
120     DISPLAY '*****'.
121     DISPLAY 'Performing post-query error checking.'.
122     PERFORM MAIN-ERROR-CHECK.
123 *
124     MAIN-ERROR-CHECK.
125     DISPLAY '*****'.
126     IF SQLCODE OF SQLCA IS EQUAL TO 0
127         DISPLAY 'No errors.'.
128     IF SQLCODE OF SQLCA IS NOT EQUAL TO 0
129         IF SQLCODE OF SQLCA IS EQUAL TO 100
130             DISPLAY 'No rows found.'
131         ELSE IF SQLCODE OF SQLCA IS EQUAL TO -213
132             DISPLAY 'Fetch interrupted by user.'
133         ELSE DISPLAY 'Error is: ', SQLCODE OF SQLCA.
```


Example Output

The output from the preceding code fragment shows how a callback routine establishes a connection to a database server, registers a callback procedure, begins SQL processing, and then terminates SQL processing when it calls the callback procedure.

```

*****
Connecting to stores7 database.
Checking for database connection errors.
*****
No errors.
*****
Registering Procedure.
Checking for registration errors.
*****
No errors.
*****
Performing long select.

Callback status code is: +0000000001

Callback procedure activated.

Attempting to determine...
if SQL processing is still active.
*****
ECO-SQD status code is: -0000000439
*****
SQL Processing is still active.
*****
Your Callback request has begun.
Please be patient.
*****

Callback status code is: +0000000002

Callback procedure activated.

Attempting to determine...
if SQL processing is still active.
*****
ECO-SQD status code is: -0000000439
*****
SQL Processing is still active.
*****
SQL Processing is still active.
Calling ECO-SQB to break SQL query.
```

```
*****
Request to break in progress.
*****
Unregistering Callback procedure.
*****
Callback procedure unregistered.
ECO-SQBCB code: +0000000000
*****
*****
Performing post-query error checking.
*****
Error is: +0000000000
*****
Disconnecting stores7 database.
Checking for disconnect errors.
*****
Error is: +0000000000
*****
Program over.
```

For more information on the syntax and use of the ECO-SQB, ECO-SQBCB, and ECO-SQD routines, refer to [“Routines That Work with the Database Server”](#) on page 5-27.

Routines That Work with the Database Server

The following sections list and fully describe the run-time routines that you can use to control the database server processes. (For additional information on connection management, refer to the description of the CONNECT, DISCONNECT, and SET CONNECTION statements in the [Informix Guide to SQL: Syntax](#).) Figure 5-7 lists the ESQL/COBOL database server manipulation routines.

Figure 5-7
Database Server Manipulation Routines

Routine Name	What It Does
ECO-SIG	Performs signal handling and cleans up child processes
ECO-SQB	Sends the database server a request to stop processing
ECO-SQBCB	Passes control to callback function during SQL processing
ECO-SQD	Checks whether database server is processing an SQL task
ECO-SQE	Terminates a database server connection
ECO-SQS	Starts a database server connection

Use these routines in your COBOL programs. When you use the **esqlcobol** compiler shell script, ESQL/COBOL automatically links run-time routines.

ECO-SIG

Purpose

Use ECO-SIG, a run-time SQL routine, to perform signal handling and cleanup for defunct child processes. When you do not clean up defunct child processes within an application, you can run out of processes.

Syntax

```
CALL ECO-SIG USING SIGMODE.
```

<i>SIGMODE</i>	a small integer set to one of the following values:
0	indicates the initial state that you must specify to enable or disable signal handling.
1	indicates that you disabled signal handling.
2	indicates that you enabled signal handling again.

Usage

If you want the Informix library to trap signals, call the ECO-SIG routine with *SIGMODE* set to 0 at the beginning of your application. This initial call to ECO-SIG must occur before the first SQL statement in the program. To turn off signal handling later, call ECO-SIG with *SIGMODE* set to 1. Then, to reenale signal handling, call ECO-SIG with *SIGMODE* set to 2.

Example

The following code fragment from the ECOSIG program shows how to use the ECO-SIG routine:

```

1  *The ECO-SIG routine enables the Informix library to perform
2  *signal handling. This program, ECOSIG, enables
3  *signal handling, disables signal handling,
4  *and re-enables signal handling.
5  *
6  IDENTIFICATION DIVISION.
7  PROGRAM-ID.
8      ECOSIG.
9  *
10 ENVIRONMENT DIVISION.
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. IFXSUN.
13 OBJECT-COMPUTER. IFXSUN.
14 *
15 DATA DIVISION.
16 WORKING-STORAGE SECTION.
17 *
18 *Declare variable.
19 *
20 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
21 77 SIGMODE SQLSMINT.
22 EXEC SQL END DECLARE SECTION END-EXEC.
23 *
24 PROCEDURE DIVISION.
25 RESIDENT SECTION 1.
26 *
27 *Begin Main routine. Enable signal handling. Disable
28 *signal handling. Re-enable signal handling.
29 *
30 MAIN.
31     DISPLAY 'ENABLE SIGNAL HANDLING OF INFORMIX LIBRARY.'
32     DISPLAY 'BEFORE EXECUTING SQL STATEMENT.'
33     DISPLAY ' '.
34     MOVE 0 TO SIGMODE.
35     CALL ECO-SIG USING SIGMODE.
36     DISPLAY 'SIGMODE IS: ', SIGMODE.
37     DISPLAY ' '.
38     EXEC SQL CONNECT TO 'stores7' END-EXEC.
39 *
40     DISPLAY 'DISABLE SIGNAL HANDLING OF INFORMIX LIBRARY.'
41     MOVE 1 TO SIGMODE.
42     CALL ECO-SIG USING SIGMODE.
43     DISPLAY 'SIGMODE IS: ', SIGMODE.

```

```
44      DISPLAY ' '.
45  *
46      DISPLAY 'RE-ENABLE SIGNAL HANDLING OF INFORMIX LIBRARY.'
47      MOVE 2 TO SIGMODE.
48      CALL ECO-SIG USING SIGMODE.
49      DISPLAY 'SIGMODE IS: ', SIGMODE.
50      DISPLAY ' '.
51  STOP RUN.
52  *
```

Example Output

The following output from the preceding code fragment shows how the ECO-SIG routine initially enables, disables, and re-enables signal handling:

```
ENABLE SIGNAL HANDLING OF INFORMIX LIBRARY
BEFORE EXECUTING SQL STATEMENT.

SIGMODE IS: +00000
EXECUTE SQL STATEMENT

DISABLE SIGNAL HANDLING OF INFORMIX LIBRARY
SIGMODE IS: +00001

RE-ENABLE SIGNAL HANDLING OF INFORMIX LIBRARY
SIGMODE IS: +00002
```

ECO-SQB

Purpose

Use ECO-SQB, a run-time SQL routine, to send the database server a request to interrupt processing of the current query.

Syntax

```
CALL ECO-SQB.
```

Usage

The database server receives the interrupt signal and returns status and control to the application process as if the SQL statement terminated with an error condition. Use ECO-SQB only after you establish a database connection. Currently, you can call the ECO-SQB routine only within a callback procedure. For a complete explanation of callback procedures, and using the ECO-SQB routine within a callback procedure, refer to [“Using Callback Procedures” on page 5-13](#).



Warning: Some resources that SQL statements establish prior to the interrupt remain after the database server terminates SQL processing. You must decide whether to terminate cursors, databases, transactions, procedures, descriptors, and so on. You probably need to roll your work back.

Example

The following example program fragment shows how to use the ECO-SQB routine. In the callback procedure, you call the ECO-SQB routine after you determine that SQL processing is still active. For a full example of how to use the ECO-SQB routine in a program, refer to [“Using Callback Procedures” on page 5-13](#).

```
1      *
2      IF ECOSQD-STAT-CODE IS EQUAL TO -419
3          DISPLAY 'Requesting SQL processing break.'
4          CALL ECO-SQB
5          DISPLAY 'SQL Processing terminated.'
```

ECO-SQBCB

Purpose

Use ECO-SQBCB, a run-time SQL routine, to register a callback procedure.

Syntax

CALL ECO-SQBCB USING *TIME-OUT*, *CALLBACK*, *STRLEN*, *STATUS*.

<i>TIME-OUT</i>	a period of time you specify in milliseconds. When that time period expires, your callback procedure executes.
<i>CALLBACK</i>	the name of your callback procedure
<i>STRLEN</i>	the length (in characters or bytes) of your callback procedure name
<i>STATUS</i>	a small integer that the ECO-SQBCB routine sets to one of the following values: <ul style="list-style-type: none"><0 indicates an unsuccessful call to ECO-SQBCB.0 indicates a successful call to ECO-SQBCB.

Usage

An SQL statement can take a long time to execute (for example, a complicated SELECT statement on a large table). Situations can occur where you want to check on the status of SQL processing or cancel SQL processing. The ECO-SQBCB routine registers a callback procedure so that, at specific time intervals while the database server processes an SQL task, the callback procedure executes and allows you to make decisions or perform other tasks without interrupting SQL processing. In addition, after your program registers the callback procedure, your program always calls the callback procedure at least twice (once at the start of SQL processing and once at the end of SQL processing.). Sometimes, SQL processing ends before your program reaches the first *TIME-OUT* interval. Also, INSERT statements stop executing when your program calls a callback procedure. You can use the callback procedure to check the status of SQL processing using the ECO-SQD routine, or to interrupt SQL processing using the ECO-SQB routine.

You unregister a callback procedure when you want the calling program to stop calling the callback procedure. After your callback procedure calls the ECO-SQB routine to break SQL processing, make sure you unregister that callback procedure. To unregister a callback procedure, move a blank string into the *CALLBACK* name variable, and a value of -1 into the *TIME-OUT* variable, then call the callback procedure.

For more information on callback procedures refer to [“Using Callback Procedures” on page 5-13](#).



Warning: Do not use the ECO-SQBCB routine when your ESQL/COBOL application uses shared memory, *olipcshm*, as the *NETTYPE* to connect to an OnLine database server. Shared memory is not a true network protocol and does not handle non-blocking I/O needed to support a callback procedure. When used with shared memory, the ECO-SQBCB call appears to register the callback function (it returns zero) but during SQL requests, the callback procedure is never called.

Examples

The following example program fragment, from the CAN-QRY program, shows how to *register* the ECO-SQBCB routine.

```

1      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
2      77  CB-STAT-CODE      PIC S9(9) COMP-5.
3      77  CB-TIME          PIC S9(9) COMP-5 VALUE 0.
4      77  CB-NAME          PIC X(12) VALUE 'CALLFUNC'.
5      77  CB-NAME-LEN      PIC S9(9) COMP-5 VALUE 12.
6      77  ECOSQB-STAT-CODE PIC S9(9) COMP-5.
7      EXEC SQL END DECLARE SECTION END-EXEC.
8      *
9      PROCEDURE DIVISION.
10     RESIDENT SECTION 1.
11     *
12     MAIN.
13 .
14 .
15 .
16     REGISTER-PROC.
17         DISPLAY ' '.
18         DISPLAY 'Registering Procedure.'.
19         CALL ECO-SQBCB USING CB-TIME, CB-NAME,
20             CB-NAME-LEN, CB-STAT-CODE.
```

The following example program fragment, from the CALLPROC program, shows how to *unregister* the ECO-SQBCB routine.

```

21     MOVE " " TO CB-NAME.
22     MOVE -1 TO CB-TIME.
23     CALL ECO-SQBCB USING CB-TIME, CB-NAME,
24         CB-NAME-LEN, CB-STAT-CODE.
```

ECO-SQD

Purpose

Use ECO-SQD, a run-time SQL routine, to check whether the database server is processing an SQL task.

Syntax

```
CALL ECO-SQD USING STATUS.
```

<i>STATUS</i>	a small integer that ECO-SQD sets to one of the following values:
0	indicates database server not currently processing an SQL task
-439	indicates database sever currently processing an SQL task

Usage

The ECO-SQD routine determines whether the database server is still processing an SQL statement. ECO-SQD returns a status code value that tells you whether the database server finished SQL processing. You use the ECO-SQD routine within a callback procedure registered using the ECO-SQBCB routine. When the *STATUS* code indicates that the database server is currently processing an SQL task, your callback procedure can call ECO-SQB to interrupt SQL processing. For more information on callback procedures refer to [“Using Callback Procedures” on page 5-13](#).

Example

The following example program code fragment shows how to use the ECO-SQD routine. In the callback procedure, you call the ECO-SQD routine to check whether the database server is currently performing SQL processing. The ECO-SQD routine returns a status code. When the status code equals -439, SQL processing is still active and the callback procedure calls the ECO-SQB routine. The ECO-SQB routine terminates SQL processing. When the status code returns 0, SQL processing is not active and the callback procedure does not call ECO-SQB.

```

1      *
2      CALL ECO-SQD USING ECOSQD-STAT-CODE.
3      DISPLAY 'Status code is: ', ECOSQD-STAT-CODE.
4      IF ECOSQD-STAT-CODE IS EQUAL TO -419
5          DISPLAY 'Requesting SQL processing break.'
6          CALL ECO-SQB
7          DISPLAY 'SQL Processing cancelled in progress.'
8      ELSE
9          DISPLAY ' '
10         DISPLAY 'SQL Processing completed.'.
11     *
```

ECO-SQE

Purpose

Use ECO-SQE, a run-time SQL routine, to terminate all database server connections, thereby freeing resources. You can use ECO-SQE 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

```
CALL ECO-SQE USING STATUS.
```

STATUS a small integer that the ECO-SQE routine sets to one of the following values:

- <0 indicates an unsuccessful call to ECO-SQE
- 0 indicates a successful call to ECO-SQE

Usage

Make sure you close all databases before you call the ECO-SQE routine. For example, before calling ECO-SQE, issue a CLOSE DATABASE statement. When you open a database that uses a transaction, and then call the ECO-SQE routine, ECO-SQE rolls back any current transactions and closes the database.

ECO-SQE behaves the same as the DISCONNECT ALL statement. However, the DISCONNECT ALL statement fails when any current transactions exist.

Example

The following code fragment from the ECOSQE program tests the ECO-SQE routine. The program connects to a database and asks you whether you want to terminate the database server connections. When you choose *y*, the program calls the ECO-SQE routine and terminates the database server process. The program attempts to disconnect from the nonexistent process, and causes an error that a GET DIAGNOSTICS statement diagnoses. The error message proves that the ECO-SQE routine terminated the connection.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOSQE.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 MESS-TEXT PIC X(254).
18 77 REPLY PIC X(1).
19 77 EX-NUM PIC S9(9) COMP-5.
20 77 COUNT-EX PIC S9(9) VALUE 1 COMP-5.
21 77 SQLSTATE PIC X(5).
22 EXEC SQL END DECLARE SECTION END-EXEC.
23 *
24 PROCEDURE DIVISION.
25 RESIDENT SECTION 1.
26 *
27 *Begin Main routine. A connection is established
28 *to stores7 database. If the user wants to terminate the
29 *database server process, the user enters 'y'. If the user
30 *does not want to terminate the database server process, the
31 *user enters any other key. If the user selected 'y', and
32 *terminated the database server process using the ECO-SQE
33 *routine, the DISCONNECT statement generates an error that
34 *verifies the ECO-SQE routine was successful in terminating
35 *the database server process. If the user selected any other
36 *key, the program disconnects without error.
37 *
```

```

38  MAIN.
39      DISPLAY 'EXECUTE CONNECT ON stores7 DATABASE'.
40      EXEC SQL
41          CONNECT TO 'stores7'
42      END-EXEC.
43      DISPLAY 'THE SQLSTATE VALUE IS: ', SQLSTATE.
44      DISPLAY 'WANT TO TERMINATE...' AT 0105.
45      DISPLAY 'DATABASE SERVER PROCESS?' AT 0205.
46      DISPLAY 'PRESS y TO TERMINATE.' AT 0305.
47      DISPLAY 'THEN PRESS RETURN.' AT 0405.
48      DISPLAY 'OTHERWISE, PRESS ANY KEY...' AT 0505.
49      DISPLAY 'THEN PRESS RETURN.' AT 0605.
50      ACCEPT REPLY AT LINE NUMBER 7 COLUMN 5
51          WITH SPACE-FILL SIZE IS 7.
52      IF REPLY IS EQUAL TO "y"
53          PERFORM CALL-ROUTINE.
54      EXEC SQL
55          DISCONNECT 'stores7'
56      END-EXEC.
57      EXEC SQL
58          GET DIAGNOSTICS :EX-NUM=NUMBER
59      END-EXEC.
60      PERFORM ERR-CHK UNTIL COUNT-EX IS GREATER THAN EX-NUM.
61  STOP RUN.
62  *
63  *Subroutine to terminate database server process.
64  *
65  CALL-ROUTINE.
66      DISPLAY ' '.
67      DISPLAY 'ATTEMPT TO DISCONNECT FROM...'.
68      DISPLAY 'CALLING ECO-SQE TO TERMINATE...'.
69      DISPLAY 'DATABASE SERVER PROCESS.'.
70      CALL ECO-SQE.
71      DISPLAY 'DATABASE SERVER PROCESS TERMINATED.'.
72  *
73  *Subroutine to diagnose and display errors.
74  *
75  ERR-CHK.
76      EXEC SQL
77          GET DIAGNOSTICS EXCEPTION :COUNT-EX
78              :SQLSTATE=RETURNED_SQLSTATE,
79              :MESS-TEXT=MESSAGE_TEXT
80      END-EXEC.
81      DISPLAY 'THE SQLSTATE VALUE IS: ', SQLSTATE.
82      DISPLAY 'THE MESSAGE TEXT IS: ', MESS-TEXT.
83      ADD 1 TO COUNT-EX.
84  *

```

Example Output

The output for the preceding code fragment shows interactive input. The output also shows the error code and error message that proves the database connection and process terminated.

```
EXECUTE CONNECT ON stores7 DATABASE
THE SQLSTATE VALUE IS: 00000
WANT TO TERMINATE THE...
DATABASE SERVER PROCESS?
PRESS y TO TERMINATE.
THEN PRESS RETURN.
OTHERWISE, PRESS ANY KEY...
THEN PRESS RETURN.
<y>
ATTEMPT TO DISCONNECT FROM...
DATABASE SERVER PROCESS.
CALLING ECO-SQE TO TERMINATE...
DATABASE SERVER PROCESS.
DATABASE SERVER PROCESS TERMINATED.
THE SQLSTATE VALUE IS: 08003
THE MESSAGE TEXT IS: Connection does not exist
```

ECO-SQS

Purpose

Use ECO-SQS, a run-time SQL routine, to *start* an implicit default connection. An implicit default connection refers to a connection to the database server that the **INFORMIXSERVER** environment variable specifies.

Syntax

```
CALL ECO-SQS.
```

Usage

You call the ECO-SQS routine only when no connections (either implicit or explicit) exist for an application. When a connection exists, ECO-SQS does nothing, even when you disconnect.

You can use the following **CONNECT** statement to establish an explicit connection to a default database server:

```
CONNECT TO DEFAULT
```

You can also use the **DATABASE** statement to establish an implicit connection to the default database server. However, the **DATABASE** statement also opens a database. For more information on explicit and implicit connections, refer to the description of the **CONNECT** statement in the [Informix Guide to SQL: Syntax](#).

ESQL/COBOL provides the ECO-SQS function for backward compatibility. Informix encourages you to use the **CONNECT** statement to establish connections.

Example

The following code fragment from the ECOSQS program tests the ECO-SQS routine. This example starts a database server connection using ECO-SQS. The program attempts to start a new database connection using a CONNECT statement. The GET DIAGNOSTICS statement diagnoses the resulting error and proves that ECO-SQS already started a database server connection.

```

1  *
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID.
4      ECOSQS.
5  *
6  ENVIRONMENT DIVISION.
7  CONFIGURATION SECTION.
8  SOURCE-COMPUTER. IFXSUN.
9  OBJECT-COMPUTER. IFXSUN.
10 *
11 DATA DIVISION.
12 WORKING-STORAGE SECTION.
13 *
14 *Declare variables.
15 *
16 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
17 77 MESS-TEXT PIC X(254).
18 77 REPLY PIC X(1).
19 EXEC SQL END DECLARE SECTION END-EXEC.
20 *
21 PROCEDURE DIVISION.
22 RESIDENT SECTION 1.
23 *
24 *Begin Main routine. Accept user input.
25 *'y' specifies that the user wants to start a
26 *database server process and the MAIN routine
27 *calls the ECO-SQS routine to start a database
28 *server process. 'n' specifies that no database
29 *server process is specified. A CONNECT is
30 *attempted. If a database server process already
31 *exists, an error results. Otherwise, there is
32 *no error.
33 *
34 MAIN.
35     DISPLAY 'DO YOU WANT TO START...' AT 0105.
36     DISPLAY 'A DATABASE SERVER PROCESS?' AT 0205.
37     DISPLAY 'ENTER y FOR YES.' AT 0305.
38     DISPLAY 'OTHERWISE, PRESS ANY OTHER KEY.' AT 0405.
39     DISPLAY 'THEN PRESS RETURN.' AT 0505.
40     ACCEPT REPLY AT LINE NUMBER6 COLUMN 5

```

```

41         WITH SPACE-FILL SIZE IS 7.
42     IF REPLY IS EQUAL TO "y"
43         CALL ECO-SQS.
44         DISPLAY ' '.
45     EXEC SQL
46         CONNECT TO DEFAULT
47     END-EXEC.
48     DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
49     EXEC SQL
50         GET DIAGNOSTICS EXCEPTION 1 :MESS-TEXT=MESSAGE_TEXT
51     END-EXEC.
52     DISPLAY 'THE ERROR MESSAGE IS: ', MESS-TEXT.
53 STOP RUN.
54 *
```

Example Output

The following output for the preceding code fragment shows interactive input. The output also shows the error code and error message related to the **CONNECT** statement.

```

DO YOU WANT TO START...
A DATABASE SERVER PROCESS ?
ENTER y FOR YES.
OTHERWISE, PRESS ANY OTHER KEY
THEN PRESS RETURN.
<y>
THE SQLSTATE CODE IS: 08002
THE ERROR MESSAGE IS: Connection name in use
```


Dynamic Management in INFORMIX-ESQL/COBOL

Programming with Dynamic SQL Statements	6-4
Working with a System Descriptor Area in INFORMIX-ESQL/COBOL	6-5
Dynamic SQL Statements and Management Techniques.	6-7
When You Need Dynamic SQL Statements	6-8
The System Descriptor Area in ESQL/COBOL	6-10
Using a System Descriptor Area	6-10
Understanding System Descriptor Area Fields.	6-12
Using Data Type Values	6-13
Using Statement Type Values	6-16
SELECT Statements That Receive WHERE-Clause Values at Run Time.	6-19
Using Host Variables	6-20
Using a System Descriptor Area	6-21
SELECT Statements in Which Select-List Values Are Determined at Run Time.	6-24
Non-SELECT Statements That Receive Values at Run Time	6-26
Using Host Variables	6-26
Using a System Descriptor Area	6-27
Non-SELECT Statements That Do Not Receive Values at Run Time	6-27
Using the EXECUTE IMMEDIATE Statement	6-28
Executing Stored Procedures That Receive Arguments at Run Time	6-29
Creating a Stored Procedure	6-30
Executing a Stored Procedure Within Your ESQL/COBOL Application.	6-30

Dynamic SQL Program Examples	6-34
The DEMO2.ECO Program	6-35
Explanation of DEMO2.ECO	6-39
The DEMO3.ECO Program	6-49
Explanation of DEMO3.ECO	6-54

Dynamic management in INFORMIX-ESQL/COBOL involves the use of dynamic SQL statements. You do not write a dynamic SQL statement as part of your program. Instead, you supply a dynamic SQL statement to your program when you execute that program.

The *[Informix Guide to SQL: Syntax](#)* illustrates and describes the following SQL statements used for dynamic management in ESQL/COBOL:

- ALLOCATE DESCRIPTOR
- FREE
- DEALLOCATE DESCRIPTOR
- GET DESCRIPTOR
- DECLARE
- OPEN
- DESCRIBE
- PREPARE
- EXECUTE
- PUT
- EXECUTE IMMEDIATE
- SET DESCRIPTOR
- FETCH

Programming with dynamic SQL in ESQL/COBOL requires that you use a system descriptor area described in “*[The System Descriptor Area in ESQL/COBOL](#)*” on page 6-10. Also refer to the discussion of dynamic SQL and other aspects of using SQL in programs in the *[Informix Guide to SQL: Tutorial](#)*.

This chapter discusses general concepts of dynamic management and includes two annotated example programs that use dynamic SQL statements.

Programming with Dynamic SQL Statements

Normally, you embed explicit SQL statements in your ESQL/COBOL program to perform predetermined activities on your database. However, advanced applications or instances could exist that do not know the precise SQL statement at compile time, as described in the following examples:

- Interactive programs, where you enter a query from the keyboard at run time
- Programs intended to work with different databases whose structures can vary

In such situations, you must work with dynamically defined SQL statements (also known as dynamic management statements). The [Informix Guide to SQL: Syntax](#) illustrates these kinds of SQL statements, that are outlined later in this chapter.

In an ESQL/COBOL program, an SQL statement passed to the PREPARE statement at run time cannot refer directly to host variables because the program has already been compiled. You must, therefore, insert a question mark (?) wherever you intended to put a host variable. These question marks indicate the parameters of the statement(s). For more information on host variables, refer to [“Using Host Variables in SQL Statements”](#) on page 1-21 and [“Choosing Data Types for Host Variables”](#) on page 2-4.

Your database server parses INFORMIX-ESQL/COBOL statements at run time. When your program uses the same statement many times, dynamic SQL statements allow you to prepare the statement once and then execute it with the values that change at run time.

The following ESQL/COBOL code fragment prepares a DELETE statement and executes that statement until C-NUM equals zero:

```
WORKING-STORAGE SECTION.  
    EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
77    C-NUM          PIC S9(4).  
..  
    EXEC SQL END DECLARE SECTION END-EXEC.  
..  
    EXEC SQL  
        PREPARE CUST_ID FROM  
            'DELETE FROM CUSTOMER WHERE CUSTOMER_NUM = ?'  
    ,  
    END-EXEC.
```



```
        DISPLAY "ENTER CUSTOMER NUMBER OR 0 TO EXIT".  
        ACCEPT C-NUM.  
        PERFORM PROCESS-DELETE UNTIL C-  
NUM IS EQUAL TO ZERO.  
    ...  
PROCESS-DELETE.  
    EXEC SQL  
        EXECUTE CUST_ID USING :C-NUM  
    END-EXEC.  
    DISPLAY "ENTER CUSTOMER NUMBER".  
    ACCEPT C-NUM.
```

In the preceding code fragment, the program evaluates the prepared statement only once, outside the performed paragraph. For an alternative, do not prepare any statement and put the DELETE statement in the performed paragraph. When the performed paragraph contains a DELETE statement, the database server parses the DELETE statement each time the database server encounters that statement.

You cannot execute SELECT statements. The DECLARE statement uses the statement *identifier* from the PREPARE statement to associate a cursor with the prepared SELECT statement. Use the USING option of the OPEN statement to communicate parameters for the SELECT statement.

Working with a System Descriptor Area in INFORMIX-ESQL/COBOL

Various dynamic SQL statements let you allocate space in memory using a system descriptor area.

Use the DESCRIBE statement to obtain information about the resulting columns in a cursor specification. The DESCRIBE statement lets you determine at run time the type of prepared statement and the number and types of data that the prepared query returns when executed.

The ALLOCATE DESCRIPTOR statement lets you allocate a system descriptor area and specify its size. Use the DEALLOCATE DESCRIPTOR statement to release the memory associated with a system descriptor area.

You can assign values to a system descriptor and retrieve the information stored in a system descriptor area through the SET DESCRIPTOR and GET DESCRIPTOR statements, respectively. You must declare the host variables used for the GET DESCRIPTOR and SET DESCRIPTOR statements in the BEGIN DECLARE SECTION of an ESQL/COBOL program with the predefined data type declarations shown in Figure 6-1.

Figure 6-1
*Predefined Data Types and Corresponding
COBOL Declarations*

Predefined Data Type	COBOL Data Type
SQLCHAR(<i>n</i>)	PIC X(10)
SQLNCHAR(<i>n</i>)	PIC X(10)
SQLINT	PIC S9(9) USAGE COMP
SQLSMINT	PIC S9(4) USAGE COMP
SQLDATE	PIC S9(9) USAGE COMP
SQLDECIMAL(<i>p,n</i>) *	PIC S9(<i>m</i>)V9(<i>n</i>) USAGE COMP
SQLMONEY(<i>p,n</i>) *	PIC S9(<i>m</i>)V9(<i>n</i>) USAGE COMP
DATE_TYPE	PIC X(10)
VARCHAR(<i>n</i>)	PIC X(<i>n</i>)
NVARCHAR(<i>n</i>)	PIC X(<i>n</i>)
FILE(<i>n</i>)	PIC X(<i>n</i>)
*(<i>m</i> , in PIC S9(<i>m</i>), equals <i>p-n</i>)	

For example, to store an integer TYPE value from the system descriptor area in a host variable, you must declare the host value as a corresponding COBOL integer, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 VAR-1 PIC S9(9) COMP-5.
EXEC SQL END DECLARE SECTION END-EXEC.
```

After you allocate a descriptor, you could set the system descriptor TYPE value of an integer as shown in the following example:

```
EXEC SQL SET DESCRIPTOR 'spdesc' VALUE 2 TYPE = 5 END-EXEC.
```

Then, to use the *VAR-1* host variable in a GET DESCRIPTOR statement, you could use the following syntax to put the TYPE value into *VAR-1*:

```
EXEC SQL GET DESCRIPTOR 'spdesc' VALUE 2  
:VAR-1 = TYPE END-EXEC.
```

The following list contains additional usage notes for the preceding data types:

- Use SQLDATE to store Julian values. Use DATE_TYPE to hold date values of the format *mm/dd/yyyy*.
- Use FILE to hold the name of the file into that you load or store a TEXT or BYTE column.
- For MF COBOL/2, substitute COMP-5 for COMP. For RM/COBOL-85, substitute COMP-1 for COMP.

You can also use a system descriptor area to provide a storage area for values returned from a FETCH statement. For more information on system descriptor fields and values, refer to [“The System Descriptor Area in ESQL/COBOL” on page 6-10](#). For more information on the ALLOCATE DESCRIPTOR, SET DESCRIPTOR, and GET DESCRIPTOR statements, refer to the [Informix Guide to SQL: Syntax](#).

Dynamic SQL Statements and Management Techniques

The following list outlines the basic process for using dynamic SQL statements in your ESQL/COBOL program:

1. Your ESQL/COBOL program assembles the text of an SQL statement as a character string in a program variable.
2. Your program executes a PREPARE statement that asks the database server to examine the statement text and prepare it for execution.
3. Your program executes the prepared statement using the EXECUTE statement.
4. Your program uses the FREE statement to explicitly free resources that the prepared statement identifier holds.

Optionally, you can use a DESCRIBE statement on any prepared statement to determine the type of statement you prepared. The following prepared outcomes can occur after a DESCRIBE statement:

- If the value in SQLCODE OF SQLCA equals 0, a SELECT statement without an INTO TEMP clause is prepared.
- If the value of SQLCODE OF SQLCA equals a positive number, some other type of statement is prepared.

To determine the type of statement you prepared, you can test the value of SQLCODE OF SQLCA against a set of predefined integers that identify statement types. Refer to [“Using Statement Type Values” on page 6-16](#) for more information about using the defined constants.

When You Need Dynamic SQL Statements

Dynamic SQL statements, more complex than nondynamic SQL statements, possess special requirements. Under the following circumstances, you must use a system descriptor area when you work with dynamically defined SQL statements in ESQL/COBOL:

- In a SELECT statement that requires input at run time to provide information for the WHERE clause
Either you know or do not know the number or data type of the parameters in the WHERE clause. For information on this type of statement, sometimes known as a parameterized SELECT statement, refer to [“SELECT Statements That Receive WHERE-Clause Values at Run Time” on page 6-19](#).
- In a SELECT statement where you do not know either the number or the data types of the columns or expressions in the *select list*, but you do know that the SELECT statement does contain a WHERE clause
For information on this type of statement, sometimes known as a non-parameterized SELECT statement, refer to [“SELECT Statements in Which Select-List Values Are Determined at Run Time” on page 6-24](#).

- In a statement other than SELECT, such as INSERT, where you do not know the number or data type of the input parameters

For information on this type of statement, sometimes known as a parameterized non-SELECT statement, refer to [“Non-SELECT Statements That Receive Values at Run Time” on page 6-26](#).

- Sometimes a statement matches these conditions:

- Not a SELECT statement
- Uses no input parameters
- Not known until run time

For information on this type of statement, sometimes known as a non-parameterized non-SELECT statement, refer to [“Non-SELECT Statements That Do Not Receive Values at Run Time” on page 6-27](#).

The first three statements in the preceding list *require* you to manage memory space for variables at run time.

If you use any of the four preceding types of statements, you must understand the concepts regarding the system descriptor area. You can use the system descriptor area to hold dynamic information. When you use a system descriptor area, you implement a language-independent method of dynamic management that makes use of the ALLOCATE DESCRIPTOR, GET DESCRIPTOR, and SET DESCRIPTOR statements in SQL. For more information about the system descriptor area, refer to [“The System Descriptor Area in ESQL/COBOL” on page 6-10](#).



Tip: *If you do not use any of the four preceding types of statements, you can skip the rest of this chapter.*

The System Descriptor Area in ESQL/COBOL

In ESQL/COBOL, you can allocate memory dynamically using a system descriptor area.

You use the system descriptor area when you use the `ALLOCATE DESCRIPTOR`, `GET DESCRIPTOR`, and `SET DESCRIPTOR` statements. These statements, described in the following list, let you determine the contents of a prepared statement at run time and allocate memory dynamically. They also let you create `WHERE` clauses for statements that receive `WHERE`-clause values at run time.

- The `ALLOCATE DESCRIPTOR` statement allocates memory for a system descriptor area that a descriptor identifies. It creates a place in memory to hold information obtained by a `DESCRIBE` statement or information about the `WHERE` clause of a statement. (The `DEALLOCATE DESCRIPTOR` statement frees the allocated system descriptor area.)
- The `GET DESCRIPTOR` statement allows you to determine how many values were described in a system descriptor area, determine the characteristics of each column or expression described in the system descriptor area, or copy a value out of the system descriptor area and into a host variable after a `FETCH` statement.
- The `SET DESCRIPTOR` statement assigns values to a system descriptor area that a descriptor identifies.

The `DESCRIBE` statement returns information about a prepared statement before you execute it. You store that information in a system descriptor area.

Using a System Descriptor Area

Dynamic SQL lets you allocate space in memory with a system descriptor area. Thus, you can use a system descriptor area and write code that supports X/Open standards.

You can allocate a system descriptor area that a *descriptor* or *descriptor variable* identifies and specify its size with the `ALLOCATE DESCRIPTOR` statement. You can use only system descriptor areas that were allocated with the `ALLOCATE DESCRIPTOR` statement in a `DESCRIBE` statement in ESQL/COBOL.

INFORMIX-ESQL/COBOL lets you take the following actions:

- Direct the output of a DESCRIBE statement on a SELECT or INSERT statement to a system descriptor area
- Set the contents of a system descriptor explicitly
- Retrieve information stored in such system descriptor areas, execute a GET DESCRIPTOR statement following a DESCRIBE statement

Use the SET DESCRIPTOR statement to assign values to an allocated system descriptor area. The DESCRIBE and SET DESCRIPTOR statements automatically allocate space for the DATA field of the system descriptor area.

You can use the system descriptor area to provide a storage area for values returned from a FETCH statement. Release memory associated with the system descriptor area with the DEALLOCATE DESCRIPTOR statement.

Other statements that support the use of a system descriptor area include EXECUTE, OPEN, and PUT. For more information on dynamic SQL and system descriptors, refer to the discussion of ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR, DESCRIBE, FETCH, EXECUTE, OPEN, and PUT in the *Informix Guide to SQL: Syntax*.

A system descriptor area has a field for the *count* of values returned by a SELECT statement or inserted into an INSERT statement. It also has a set of fields for each value or item entered or returned. Figure 6-2 illustrates a descriptor area for two values.

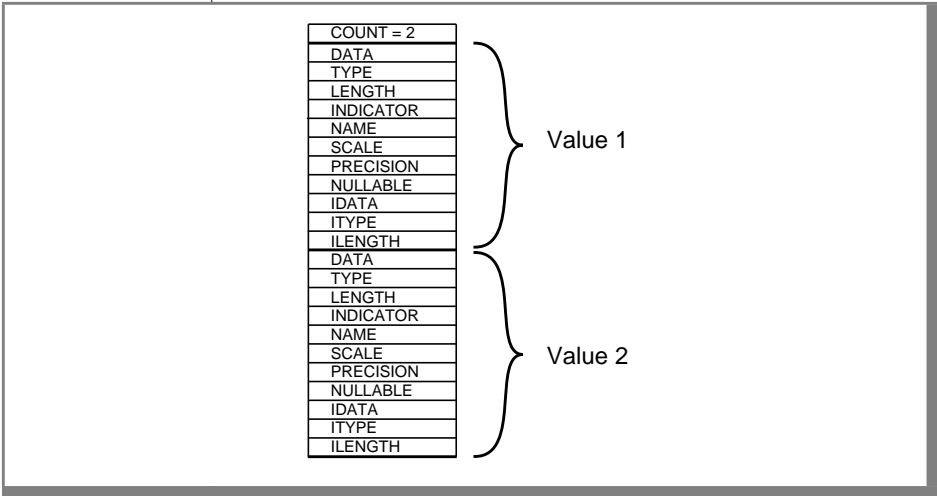


Figure 6-2
A System Descriptor Area for Two Values

Understanding System Descriptor Area Fields

The following alphabetically ordered list describes the standard fields of the system descriptor area:

COUNT	represents the number of VALUES, items, or <i>occurrences</i> in the system descriptor area. For example, after you set ALLOCATE DESCRIPTOR, COUNT holds number of occurrences. DESCRIBE sets COUNT to the number of values in the SELECT or INSERT list. (You can obtain this number using GET DESCRIPTOR.). When you use a system descriptor area to hold parameters for a PUT, OPEN, or EXECUTE statement, you must set the COUNT field to the number of parameters.
DATA	represents the data. DATA can represent a host variable or a numeric literal, character string literal, DATETIME literal, or INTERVAL literal. This field does not represent a standard X/Open field and makes a warning message appear in X/Open mode.
IDATA	represents the user-defined indicator data or the name of a host variable that contains indicator data for the DATA field. This field does not represent a standard X/Open field and makes a warning message appear in X/Open mode.
ILENGTH	represents the length, in bytes, of the user-defined indicator. This field does not represent a standard X/Open field and makes a warning message appear in X/Open mode.
INDICATOR	represents a short integer indicator variable. INDICATOR can contain two values: 0 means non-null data exists in the DATA field, and -1 means null data exists in the DATA field.
ITYPE	represents the data type for a user-defined, short-integer indicator. Figure 6-3 on page 6-14 and Figure 6-4 on page 6-14 define the integer correspondences.
LENGTH	represents a short integer that gives the size in bytes of CHAR type data, the encoded qualifiers of DATETIME or INTERVAL data, or the size of a DECIMAL or MONEY value.

NAME	represents a character string containing the column name or display label you transfer.
NULLABLE	<p>specifies whether a resulting column can contain a null value after a you execute a DESCRIBE statement. The value 1 means the column allows null values, and the value 0 means the column does <i>not</i> allow null values.</p> <p>Before you execute an EXECUTE statement or a dynamic OPEN statement, you must set NULLABLE to 1 to indicate that the INDICATOR field specifies an indicator value, and to 0 when you do not specify an indicator value. (When executing a dynamic FETCH statement, your ESQL/COBOL program ignores the NULLABLE field.)</p>
PRECISION	you define this field only for the DECIMAL or MONEY data type. After you execute a DESCRIBE statement, PRECISION contains the precision of the column. Otherwise, you must set PRECISION to indicate the precision of the value in the DATA field.
SCALE	you define this field only for the DECIMAL or MONEY data type. After you execute a DESCRIBE statement, SCALE contains the scale of the column. In a SET DESCRIPTOR statement, you must set SCALE to indicate the scale of the value in the DATA field.
TYPE	represents a short integer corresponding to the data type you transfer. Figure 6-3 and Figure 6-4 define the integer correspondences.

Using Data Type Values

Within an INFORMIX-ESQL/COBOL program that uses dynamically defined SQL statements, you can use the constant integer values shown in Figure 6-3 and Figure 6-4. Use these language-independent constants when you analyze the information a DESCRIBE statement returns to a system descriptor area or when you set the TYPE in a SET DESCRIPTOR statement.

Figure 6-3 and Figure 6-4 show the values for TYPE and ITYPE in X/Open mode and in standard mode.

Figure 6-3

Values for the TYPE and ITYPE Fields for X/Open SQL

Data Type	Integer
CHARACTER	1
DECIMAL	3
INTEGER	4
SMALLINT	5
FLOAT	6

Figure 6-4

Values for the TYPE and ITYPE Fields When Not Using X/Open SQL

Data Type	Integer
CHARACTER	0
DECIMAL	5
INTEGER	2
SMALLINT	1
FLOAT	3
SMALLFLOAT	4
SERIAL	6
DATE	7
MONEY	8
DATETIME	10
BYTE	11
TEXT	12

(1 of 2)

Data Type	Integer
VARCHAR	13
INTERVAL	14
FILE	116

(2 of 2)

The following code fragment from the SETD program shows how to set and use an INTEGER TYPE field in SET DESCRIPTOR and GET DESCRIPTOR statements:

```

1  *
2  *This program, SETD, shows how to allocate a
3  *descriptor, set a descriptor, get a descriptor
4  *and display the contents of a descriptor.
5  *
6  IDENTIFICATION DIVISION.
7  PROGRAM-ID.
8      SETD.
9  *
10 ENVIRONMENT DIVISION.
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. IFXSUN.
13 OBJECT-COMPUTER. IFXSUN.
14 *
15 DATA DIVISION.
16 WORKING-STORAGE SECTION.
17 *
18 *Display variable.
19 *
20 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
21 77 INT-TYPE PIC S9(9) COMP-5.
22 EXEC SQL END DECLARE SECTION END-EXEC.
23 *
24 PROCEDURE DIVISION.
25 RESIDENT SECTION 1.
26 *
27 *Begin Main routine. Allocate and set a
28 *descriptor. Move the contents of a
29 *descriptor into a host variable and
30 *display the host variable.
31 *
```

```

32  MAIN.
33      EXEC SQL ALLOCATE DESCRIPTOR 'desc_100'
34          WITH MAX 5 END-EXEC.
35      EXEC SQL
36          SET DESCRIPTOR 'desc_100' VALUE 2 TYPE = 5
37      END-EXEC.
38      EXEC SQL
39          GET DESCRIPTOR 'desc_100' VALUE 2
40          :INT-TYPE = TYPE
41      END EXEC.
42      DISPLAY 'The type value is: ', INT-TYPE.
43  STOP RUN.

```

Using Statement Type Values

After you use the DESCRIBE statement, the database server sets SQLCODE OF SQLCA to a positive integer value, that indicates the type of statement that was described. In other words, SQLCODE OF SQLCA indicates whether the statement was INSERT, SELECT, CREATE TABLE, or another kind of statement.

To determine the kind of statement described, check the value of SQLCODE OF SQLCA against a predefined set of values. Figure 6-5 lists the predefined integer constants for types of SQL statements.

Figure 6-5
Integer Constants for Types of SQL Statement

Statement	Value
SELECT (no INTO TEMP clause)	0
DATABASE	1
<i>Internal use only</i>	2
SELECT INTO	3
UPDATE... WHERE	4
DELETE... WHERE	5
INSERT	6
UPDATE... WHERE CURRENT OF	7
DELETE... WHERE CURRENT OF	8

(1 of 4)

Statement	Value
<i>Internal use only</i>	9
LOCK TABLE	10
UNLOCK TABLE	11
CREATE DATABASE	12
DROP DATATBASE	13
CREATE TABLE	14
DROP TABLE	15
CREATE INDEX	16
DROP INDEX	17
GRANT FRAGMENT	18
REVOKE FRAGMENT	19
RENAME TABLE	20
RENAME COLUMN	21
CREATE AUDIT	22
DROP AUDIT	25
RECOVER TABLE	26
<i>Internal use only</i>	27-28
ALTER TABLE	29
UPDATE STATISTICS	30
CLOSE DATABASE	31
DELETE (no WHERE clause)	32
UPDATE (no WHERE clause)	33
BEGIN WORK	34
COMMIT WORK	35

(2 of 4)

Statement	Value
ROLLBACK WORK	36
<i>Internal use only</i>	37
START DATABASE	38
ROLL FORWARD	39
CREATE VIEW	40
DROP VIEW	41
<i>Internal use only</i>	42
CREATE SYNONYM	43
DROP SYNONYM	44
CREATE TEMP TABLE	45
SET LOCK MODE	46
ALTER INDEX	47
SET ISOLATION, SET TRANSACTION	48
SET LOG	49
SET EXPLAIN	50
CREATE SCHEMA	51
SET OPTIMIZATION	52
CREATE PROCEDURE	53
DROP PROCEDURE	54
SET CONSTRAINTS	55
EXECUTE PROCEDURE	56
SET DEBUG FILE TO	57
CREATE OPTICAL CLUSTER	58
ALTER OPTICAL CLUSTER	59

(3 of 4)

Statement	Value
DROP OPTICAL CLUSTER	60
RESERVE (optical)	61
RELEASE (optical)	62
SET MOUNTING TIMEOUT	63
UPDATE STATISTICS... for procedure	64
<i>Defined for Kanji version only</i>	65-66
<i>Reserved</i>	67-69
CREATE TRIGGER	70
DROP TRIGGER	71
<i>Reserved</i>	72
SET	76
START VIOLATIONS TABLE	77
STOP VIOLATIONS TABLE	78

(4 of 4)

SELECT Statements That Receive WHERE-Clause Values at Run Time

In SELECT statements that receive WHERE-clause values at run time, you do not know the number or data type of the parameters in the WHERE clause. You must supply the input variables at run time. Because DESCRIBE statements examine only the list of column names or expressions in the SELECT statement, they do not tell you about parameters in the WHERE clauses.

You must know the number of parameters in the SELECT statement and their data types. Unless you are writing a general-purpose, interactive interpreter, you usually know this information. When you do not know it, you must write code that determines not only how many question marks (?) appear in the dynamic query but also to what data type they belong.

When you know the number of parameters and their data types at compile time, you can declare appropriate host variables to receive the parameter values and run the query using those values.

When you determine the number of parameters and their data types at compile time, you use a system descriptor area to pass data to the query.

Using Host Variables

Perform the following steps to use host variables with a SELECT statement that receives WHERE-clause values at run time:

1. Declare a host variable for each parameter in the WHERE clause of the SELECT statement.
2. Prepare the SELECT statement. It must contain a question mark (?) for each missing value in the WHERE clause.
3. Use the DECLARE statement to associate a cursor with the prepared SELECT statement.
4. Assign a value to the host variable for each parameter. (Usually, the application obtains these values interactively.)
5. Use the OPEN statement with the USING clause to associate the host variables (and their contents) with the question marks (?) in the prepared SELECT statement.
6. Use the FETCH statement to get the first set of values that the prepared SELECT returns. Repeat the fetch until the database server returns no more rows.
7. Close the cursor using the CLOSE statement.

In the following example, the host variables that correspond to the parameters in the SELECT statement include only HOSTVAR1, HOSTVAR2, and HOSTVAR3. Execute the OPEN statement as shown in this example:

```
EXEC SQL INCLUDE SQLCA END-EXEC.  
*  
* DECLARE PARAMETER VARIABLES  
  
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
    05 HOSTVAR1 PIC S9(9) COMP-5.  
    05 HOSTVAR2 PIC S9(9) COMP-5.  
    05 HOSTVAR3 PIC S9(9) COMP-5.  
EXEC SQL END DECLARE SECTION END-EXEC.
```



```
* MAKE STORES7 THE CURRENT DATABASE

EXEC SQL CONNECT TO 'STORES7' END-EXEC.

* PREPARE THE SELECT STATEMENT

EXEC SQL PREPARE Q_ID FROM
'SELECT ORDER_NUM, CUSTOMER_NUM
  FROM ORDERS
  WHERE ORDER_DATE = ?
        OR PAID_DATE = ?
        OR SHIP_DATE = ?'
END-EXEC.

* ASSOCIATE Q_CURSOR WITH SELECT STATEMENT

EXEC SQL DECLARE Q_CURSOR CURSOR FOR Q_ID END-EXEC.

* THIS SECTION OF THE PROGRAM WOULD
* ASSIGN VALUES TO HOSTVAR1, HOSTVAR2, AND HOSTVAR3

* OPEN Q_CURSOR TO CREATE THE ACTIVE SET

EXEC SQL
  OPEN Q_CURSOR USING :HOSTVAR1, :HOSTVAR2, :HOSTVAR3
END-EXEC.
```

Using a System Descriptor Area

Your code must include the following steps to handle a SELECT statement that receives WHERE-clause values at run time. Consider the first four steps common to all SELECT statements whether or not they contain a WHERE clause.

1. Declare host variables to hold the data obtained from the user interactively.
2. Prepare the SELECT statement (using the PREPARE statement), and give it a statement identifier. The SELECT statement must contain a question mark (?) for each missing value in the WHERE clause. In the following example, the statement identifier could possibly represent QID:

```
EXEC SQL
  PREPARE QID FROM
'SELECT * FROM CUSTOMER WHERE LNAME > ?'
END-EXEC.
```

3. Declare a cursor for the prepared statement identifier (for example, DEMOCURSOR). You must associate all dynamically defined SELECT statements with a declared cursor. For example, the following statement declares the cursor DEMOCURSOR for QID:

```
EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
```

4. Allocate a descriptor using the ALLOCATE DESCRIPTOR statement. Provide a name for the descriptor area and indicate the maximum number of items that can exist in the select list of the query. When you do not provide a maximum, ESQL/COBOL allocates space for 100 returned items. The following statement allocates a descriptor named DESC, that provides room for up to a four-item select list:

```
EXEC SQL
    ALLOCATE DESCRIPTOR 'DESC' WITH MAX 4
END-EXEC.
```

5. Your COBOL code must analyze the WHERE clause of the SELECT statement to determine how many and what type of parameters reside in the WHERE clause.
6. Once you determine the number of question marks in the query, you must use an ALLOCATE DESCRIPTOR statement to allocate a descriptor large enough to handle the filter variables. When you allocate a descriptor of sufficient size in Step 4, you can use that.
7. Issue a SET DESCRIPTOR statement for each of the question marks (filter values) in the SELECT statement. The SET DESCRIPTOR statement must set the TYPE and VALUE fields of the descriptor area. When you specify the TYPE field as CHAR or VARCHAR, you must also provide a value for the LENGTH field. The other fields are optional.

The following statement sets the first value in the descriptor area for a character value with a value assigned from the HOSTCHAR host variable:

```
EXEC SQL
    SET DESCRIPTOR 'DESC' VALUE 1
        TYPE = 0,
        LENGTH = 15,
        DATA = :HOSTCHAR
END-EXEC.
```



Important: When you use X/Open code (and compiling with the `-xopen` flag), use the language-independent integer values for the X/Open environment to set the TYPE field. Refer to “Using Data Type Values” on page 6-13.

8. Once you set all the necessary information for each VALUE, open a cursor using the descriptor. For the preceding SET DESCRIPTOR statement, you can use the following OPEN statement:

```
EXEC SQL
  OPEN DEMOCURSOR USING SQL DESCRIPTOR 'DESC'
END-EXEC.
```

9. Determine the contents of the select list of the query. To do this, use the DESCRIBE statement with the descriptor that you allocated for the returned values. For example, the following statement describes the prepared query QID into the DESC descriptor.

```
EXEC SQL
  DESCRIBE QID USING SQL DESCRIPTOR 'DESC'
END-EXEC.
```

10. To use the GET DESCRIPTOR statement to determine the count of values in the select list, look at the COUNT field of the descriptor. For example, the following statement puts the count of the values in the select list into the COUNT host variable.

```
EXEC SQL
  GET DESCRIPTOR 'DESC' :COUNT = COUNT
END-EXEC.
```

11. Determine the type, length, name, and other information about each of the values described into the descriptor as your program needs such information for formatting or processing. For example, to determine the type of the third value in a select list, issue the following statement:

```
EXEC SQL
  GET DESCRIPTOR 'DESC' VALUE 3 :TYPE_INT = TYPE
END-EXEC.
```

12. Fetch each row of values returned with the SELECT statement in a loop until the program finds no more rows (SQLCODE OF SQLCA = SQLNOTFOUND). After each FETCH statement, use the GET DESCRIPTOR statement on each value in the select list to load the contents of the DATA field into an appropriate host variable for your program to use. For example, the following statement copies the data for the second value into the host variable RESULT:

```
EXEC SQL
  GET DESCRIPTOR 'DESC' VALUE 2 :RESULT = DATA
END-EXEC.
```

13. After the program fetches all the rows, close the cursor using the CLOSE statement.

SELECT Statements in Which Select-List Values Are Determined at Run Time

These kinds of statements occur when you know neither the number nor the data types of the members of the *select list*, nor do you know the list of column names or expressions in the SELECT statement.

In nondynamic SELECT statements, ESQL/COBOL places the values returned from the query into host variables listed in an INTO clause. When you create a SELECT statement interactively after you compile your program, you cannot use an INTO clause because the host variables are not directly available. Instead, you must use the system descriptor area to hold the selected values. Follow these steps to program the code that uses a SELECT statement, that determines select-list values at run time:

1. Prepare the SELECT statement, using the PREPARE statement, and give it a statement identifier (for example, QID).
2. Declare a cursor for the prepared statement identifier (such as DEMOCURSOR). You must associate all dynamically defined SELECT statements with a declared cursor. For example, the following statement declares the cursor DEMOCURSOR for QID:

```
EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
```

3. Allocate a descriptor using the ALLOCATE DESCRIPTOR statement and provide a name for the descriptor area. Use the WITH MAX clause to indicate the maximum number of items that can reside in the select list of the query. When you do not provide a maximum, ESQL/COBOL allocates space for 100 returned items. For example, the following statement allocates a descriptor named DESC, that provides room for up to a four-item select list:

```
EXEC SQL  
    ALLOCATE DESCRIPTOR 'DESC' WITH MAX 4  
END-EXEC.
```

4. Open the cursor, as shown in the following example:

```
EXEC SQL OPEN DEMOCURSOR END-EXEC.
```

Important: If the SELECT statement has a WHERE clause, your program must also handle the WHERE clause. Refer to page 6-19 for information on handling SELECT statements that receive WHERE-clause values at run time.



5. Determine the contents of the select list of the query. To do this, use the DESCRIBE statement with the descriptor that you allocated for the returned values. For example, the following statement describes the prepared query QID into the DESC descriptor:

```
EXEC SQL
    DESCRIBE QID USING SQL DESCRIPTOR 'DESC'
END-EXEC.
```

6. To use the GET DESCRIPTOR statement to determine the count of values in the select list, look at the COUNT field of the descriptor. For example, the following statement puts the count of the values in the select list into the COUNT host variable:

```
EXEC SQL
    GET DESCRIPTOR 'DESC' :COUNT = COUNT
END-EXEC.
```

7. Determine the type, length, name, and other information about each of the values described into the descriptor as your program needs such information for formatting or processing. For example, to determine the type of the third value in a select list, issue the following statement:

```
EXEC SQL
    GET DESCRIPTOR 'DESC' VALUE 3 :TYPE_INT = TYPE
END-EXEC.
```

8. Fetch each row of values returned with the SELECT statement in a loop until the program finds no more rows (SQLCODE OF SQLCA = SQLNOTFOUND). After each FETCH statement, use the GET DESCRIPTOR statement on each value in the select list to load the contents of the DATA field into an appropriate host variable for your program to use. For example, the following statement copies the data for the second value into the host variable RESULT:

```
EXEC SQL
    GET DESCRIPTOR 'DESC' VALUE 2 :RESULT = DATA
END-EXEC.
```

9. After the program fetches all the rows, close the cursor using the CLOSE statement.

Non-SELECT Statements That Receive Values at Run Time

ESQL/COBOL treats non-SELECT statements that possess an unknown number or data type of the input parameters essentially the same as SELECT statements that receive WHERE-clause values at run time. One difference exists in that you use the EXECUTE statement rather than the OPEN statement to indicate the parameters.

The DELETE and UPDATE statements can both contain a WHERE clause. Although similar to using a SELECT with a WHERE clause, your program associates the parameters to the prepared statement in an EXECUTE statement rather than in an OPEN statement. When you dynamically use a DELETE or UPDATE statement, your program must follow the steps given in the following list:

1. Prepare the dynamic non-SELECT statement using the PREPARE statement.
2. Describe the statement and check the value of SQLWARN4 in the SQLCA record. When SQLWARN4 contains a W, the DELETE or UPDATE statement does not contain a WHERE clause and, when executed, the program deletes or updates all the rows in the table.
3. Execute the prepared statement using the EXECUTE statement with the appropriate USING clause. You can use host variables or a system descriptor area to hold the parameters.

Using Host Variables

If you know the number of parameters and their data types at compile time, you can execute an SQL statement that you prepared using the names of the host variables that hold the parameter data. For example, use the following statement to run a DELETE statement or an UPDATE statement that requires three parameters, with the parameter values stored in :HOSTVAR1, :HOSTVAR2, and :HOSTVAR3, and that was prepared with the identifier STATEID:

```
EXEC SQL
    EXECUTE STATEID USING :HOSTVAR1, :HOSTVAR2, :HOSTVAR3
END-EXEC.
```

Using a System Descriptor Area

You can use a system descriptor area to hold the information about the parameters by performing the steps in the following list:

1. Allocate a descriptor large enough to hold the parameters with the `ALLOCATE DESCRIPTOR` statement.

```
EXEC SQL
    ALLOCATE DESCRIPTOR 'UP_DESC' WITH MAX 10
END-EXEC.
```

2. For each parameter, use the `SET DESCRIPTOR` statement to set the `TYPE` of each parameter and associate the host variable that holds the data with the descriptor field. The following example sets the second value in the `UP_DESC` descriptor to an integer value that receives its data from the host variable called `:H_INT`. Use this value for the second question mark (?) in the `WHERE` clause.

```
EXEC SQL
    SET DESCRIPTOR 'UP_DESC' VALUE 2
        TYPE = 2, DATA = :H_INT
END-EXEC.
```

3. Use the `EXECUTE` statement with the `USING SQL DESCRIPTOR` clause. For example, the following statement associates information about the parameters held in `UP_DESC` with the prepared statement called `STATEID`:

```
EXEC SQL
    EXECUTE STATEID USING SQL DESCRIPTOR 'UP_DESC'
END-EXEC.
```

Non-SELECT Statements That Do Not Receive Values at Run Time

In many cases, you can assemble a statement at run time using a simple process. When that statement is *not* a `SELECT` statement, and you know the basic structure of the statement and all its components when you write your program, you can simply prepare and execute the statement.

For example, you could write a general-purpose deletion program that works on any table. Your program performs the following steps:

1. Your program prompts the user for the name of the table and the text of the WHERE clause and puts the information into host variables such as :TABNAME and :SEARCH_CONDITION.
2. It concatenates four components (:DELETE FROM, :TABNAME, WHERE, and :SEARCH_CONDITION) and thus creates a text string.
3. Your program then prepares the whole statement. When your program calls the DELETE statement string :STMT_BUF and assigns it an ID of D_ID for the PREPARE statement, you use the following PREPARE statement:

```
EXEC SQL PREPARE D_ID FROM :STMT_BUF END-EXEC.
```

4. The program then executes the prepared statement, as shown in the following example:

```
EXEC SQL EXECUTE D_ID END-EXEC.
```

Using the EXECUTE IMMEDIATE Statement

Instead of preparing a statement and then executing it, you can prepare and execute the statement in the same step with the EXECUTE IMMEDIATE statement.

For example, for the DELETE statement described in the preceding section, you replace the PREPARE and EXECUTE statement sequence with the following statement:

```
EXEC SQL EXECUTE IMMEDIATE :STMT_BUF END-EXEC.
```

The EXECUTE IMMEDIATE statement also implicitly frees the memory resources that the prepared statement uses.

Executing Stored Procedures That Receive Arguments at Run Time

In SQL, a stored procedure is a user-defined function. Anyone who has a resource privilege on a database can create a stored procedure. Once you create a stored procedure, you store that procedure in an executable format in the database as an object of the database. You can use stored procedures to perform any function you can perform in SQL, and expand what you can accomplish with SQL alone.

You write a stored procedure with SQL and SPL statements. You can use only SPL statements inside CREATE PROCEDURE and CREATE PROCEDURE FROM statements. DB-Access and INFORMIX-ESQL/COBOL can use those statements. You can use DB-Access to create a stored procedure. You can also create a stored procedure at the beginning of an ESQL/COBOL program. However, using DB-Access provides the most simple way to create a stored procedure.

You can accomplish a wide range of objectives with stored procedures, including improving database performance, simplifying the writing of applications, and limiting or monitoring access to data.

INFORMIX-ESQL/COBOL allows you to call, or execute, a stored procedure and pass arguments to that procedure at run time. Because a stored procedure exists in executable format, you can use it to execute frequently repeated tasks to improve performance. Executing a stored procedure rather than straight SQL code allows you to bypass repeated parsing, validity checking, and query optimization. For further information on stored procedures, refer to the [Informix Guide to SQL: Tutorial](#).

This section provides a simple example of how to write an ESQL/COBOL program that receives a value at run-time and passes that value as an argument to a stored procedure. The stored procedure executes, adds a number to the argument, and returns the sum to the ESQL/COBOL program.

Creating a Stored Procedure

To create a stored procedure, you can use the CREATE PROCEDURE statement. Create the CREATE PROCEDURE statement in a file of your choice using your UNIX **vi** editor or any other suitable UNIX editor. Any program you write can create a stored procedure. Make your program access this file and invoke the CREATE PROCEDURE statement that resides within that file.

For the purposes of the code fragment from the PRO1 program listed on page 6-31, you name the following CREATE PROCEDURE statement **test1**. In this example, **/tmp/stevek/crpro** represents the full pathname of the file that contains the CREATE PROCEDURE statement. The stored procedure **test1** accepts one integer argument from an ESQL/COBOL program, designates the integer variable **total** to receive the sum of the integer argument and the number five, and returns the value of **total** to the ESQL/COBOL program.

```
CREATE PROCEDURE test1(a_units INT)
  RETURNING INT;
  DEFINE total INT;
  LET total = a_units + 5;
  RETURN total;
END PROCEDURE;
```

Executing a Stored Procedure Within Your ESQL/COBOL Application

The following ESQL/COBOL program dynamically allocates memory for a stored procedure, passes one argument to a stored procedure, executes the stored procedure, stores the resulting value (that the stored procedure returns) in dynamic memory, and then displays the value. The program performs all of those tasks at run time. For the purpose of simplicity, this program uses an EXECUTE PROCEDURE statement that resides in the variable **PRC**. You can also use interactive input to store an EXECUTE PROCEDURE statement in a variable. The program PRO1 takes the following actions:

1. Declares the variable **PRC** to contain an EXECUTE PROCEDURE statement. That statement calls the stored procedure **test1** and passes an integer value of 100 to that procedure.
2. Declares the variable **INT-DATA**. After the stored procedure executes, the program passes the data (that the stored procedure returns) to a dynamic memory storage area called **DATA**. **INT-DATA** receives the data value from **DATA**. Then, a display statement uses **INT-DATA**.

3. Executes the MAIN routine of the program.
4. Connects to a database server.
5. Creates a database used to contain a stored procedure.
6. Executes a CREATE PROCEDURE FROM statement that invokes the CREATE PROCEDURE statement residing in the file **/tmp/stevek/crpro**.
7. Prepares the EXECUTE PROCEDURE statement contained in the variable *PRC* and designates **prcstmt** as the prepared statement name.
8. Allocates a descriptor area named **spdesc**.
9. Describes the descriptor area into the descriptor **spdesc**.
10. Declares the cursor **spcurs** for the prepared statement.
11. Opens the cursor and thereby invokes the EXECUTE PROCEDURE statement in *PRC*.
12. Fetches the cursor using the descriptor **spdesc**.
13. Passes the value from the descriptor *DATA* into the variable *INT-DATA*.
14. Displays the integer value 105 that the stored procedure returned (100+5).
15. Closes the cursor.
16. Drops the stored procedure **test1** from the database.
17. Closes the database.
18. Drops the database.
19. Disconnects from the database server.

```
1  *
2  *This program, PR01, executes a stored procedure
3  *and retrieves the value returned by the stored
4  *procedure into a descriptor.
5  *An integer value, 100, is sent to
6  *the stored procedure at run time. The stored
7  *procedure adds 5 to 100. The resultant value,
8  *105, is displayed at the end of the program.
9  *
10 IDENTIFICATION DIVISION.
11 PROGRAM-ID.
12     PR01.
13 *
14 ENVIRONMENT DIVISION.
15 CONFIGURATION SECTION.
```

Executing a Stored Procedure Within Your ESQL/COBOL Application

```
16  SOURCE-COMPUTER. IFXSUN.
17  OBJECT-COMPUTER. IFXSUN.
18  *
19  DATA DIVISION.
20  WORKING-STORAGE SECTION.
21  *
22  *Declare variables.
23  *
24  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
25  77 SP-COUNT PIC S9(4) COMP-5.
26  77 PRC PIC X(100)
27  -      VALUE "EXECUTE PROCEDURE
28  -      test1(100)".
29  77 INT-DATA PIC S9(9) COMP-5.
30  EXEC SQL END DECLARE SECTION END-EXEC.
31  *
32  PROCEDURE DIVISION.
33  RESIDENT SECTION 1.
34  *
35  *Begin Main routine. Connect to the database
36  *server. Create a procedure from an external
37  *file. Prepare an EXECUTE PROCEDURE statement.
38  *Allocate a descriptor. Execute a DESCRIBE statement
39  *to define the returned value from the EXECUTE
40  *PROCEDURE statement in the descriptor area.
41  *DECLARE a cursor. OPEN the cursor. FETCH a value
42  *into the descriptor. GET the descriptor contents
43  *and move those contents into a host variable. Display
44  *the returned data value. CLOSE the cursor. DROP
45  *the procedure. CLOSE the database. DROP the database.
46  *Terminate the connection.
47  *
48  MAIN.
49      EXEC SQL CONNECT TO DEFAULT END-EXEC.
50      EXEC SQL CREATE DATABASE db1 END-EXEC.
51      EXEC SQL
52          CREATE PROCEDURE FROM '/tmp/stevek/crpro'
53          END-EXEC.
54      EXEC SQL PREPARE prcstmt FROM :PRC END-EXEC.
55      EXEC SQL ALLOCATE DESCRIPTOR 'spdesc'
56          WITH MAX 1 END-EXEC.
57      EXEC SQL DESCRIBE prcstmt USING SQL DESCRIPTOR
58          'spdesc' END-EXEC.
59      EXEC SQL DECLARE spcurs CURSOR FOR prcstmt
60          END-EXEC.
61      EXEC SQL OPEN spcurs END-EXEC.
62      EXEC SQL FETCH spcurs USING SQL DESCRIPTOR
63          'spdesc' END-EXEC.
64      EXEC SQL GET DESCRIPTOR 'spdesc' VALUE 1
```

```
65         :INT-DATA = DATA END-EXEC.  
66     DISPLAY 'DATA IS: ', INT-DATA.  
67     EXEC SQL CLOSE spcurs END-EXEC.  
68     EXEC SQL DROP PROCEDURE test1 END-EXEC.  
69     EXEC SQL CLOSE DATABASE END-EXEC.  
70     EXEC SQL DROP DATABASE db1 END-EXEC.  
71     EXEC SQL DISCONNECT ALL END-EXEC.  
72 STOP RUN.  
73 *
```

The following example shows the output of the PRO1 program. The stored procedure adds the values 100 and 5 and returns the sum (105), through dynamic memory, to the program.

```
The DESCRIPTOR COUNT IS: +00001  
Data is: +0000000105
```

Alternatively, you can use the EXECUTE INTO statement to replace the PREPARE, OPEN, and FETCH statements in the preceding program. The following example code fragment, from the PRO2 program, shows how to alter the PRO1 program to use the EXECUTE INTO statement.

```
1         MAIN.  
2         EXEC SQL CONNECT TO DEFAULT END-EXEC.  
3         EXEC SQL CREATE DATABASE db1 END-EXEC.  
4         EXEC SQL  
5             CREATE PROCEDURE FROM 'crpro'  
6             END-EXEC.  
7         EXEC SQL PREPARE prcstmt FROM :PRC END-EXEC.  
8         EXEC SQL ALLOCATE DESCRIPTOR 'spdesc'  
9             WITH MAX 1 END-EXEC.  
10        EXEC SQL DESCRIBE prcstmt USING SQL DESCRIPTOR  
11            'spdesc' END-EXEC.  
12        EXEC SQL  
13            EXECUTE prcstmt INTO SQL DESCRIPTOR 'spdesc'  
14            END-EXEC.  
15        EXEC SQL GET DESCRIPTOR 'spdesc' VALUE 1  
16            :INT-DATA = DATA END-EXEC.  
17        DISPLAY 'DATA IS: ', INT-DATA.  
18        EXEC SQL CLOSE spcurs END-EXEC.  
19        EXEC SQL DROP PROCEDURE test1 END-EXEC.  
20        EXEC SQL CLOSE DATABASE END-EXEC.  
21        EXEC SQL DROP DATABASE db1 END-EXEC.  
22        EXEC SQL DISCONNECT ALL END-EXEC.  
23 STOP RUN.
```

Dynamic SQL Program Examples

Your INFORMIX-ESQL/COBOL software includes the DEMO2.ECO and DEMO3.ECO programs. You can create those programs when you respond affirmatively to the **esqlcobdemo7** prompt as discussed in “[Demonstration Database](#)” in the Introduction.

The first example, DEMO2.ECO, functionally similar to the DEMO1.ECO example shown in [Chapter 1, “Programming with INFORMIX-ESQL/COBOL,”](#) declares the cursor for a prepared statement with an unknown value. The program opens the cursor and supplies the value, fetches the data, and closes the cursor.

The second example, DEMO3.ECO, was designed as a modified version of the DEMO2.ECO example. DEMO3.ECO also uses a PREPARE statement on a query with an unknown parameter in the WHERE clause and includes the DESCRIBE, GET DESCRIPTOR, ALLOCATE DESCRIPTOR, and DEALLOCATE DESCRIPTOR statements.

Both examples use the SQLSTATE value not only to detect errors but also to signal the end of the active list of rows that the SELECT statement returned.

The DEMO2.ECO Program

The DEMO2.ECO program uses a PREPARE statement on a query with an unknown parameter in the WHERE clause.

```

24 *
25 *This program, DEMO2, declares a cursor for a
26 *prepared statement with an unknown value. The
27 *value is supplied when the cursor is opened,
28 *the data is fetched, and the cursor is closed.
29 *Display the data (names).
30 *
31 IDENTIFICATION DIVISION.
32 PROGRAM-ID.
33     DEMO2.
34 *
35 ENVIRONMENT DIVISION.
36 CONFIGURATION SECTION.
37 SOURCE-COMPUTER. IFXSUN.
38 OBJECT-COMPUTER. IFXSUN.
39 *
40 *Declare variables.
41 *
42 DATA DIVISION.
43 WORKING-STORAGE SECTION.
44 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
45 77 FNAME          PIC X(15).
46 77 LNAME          PIC X(20).
47 77 DEMO-QUERY     PIC X(50)
48     VALUE "SELECT FNAME, LNAME FROM CUSTOMER WHERE LNAME>?".
49 77 QUERY-VALUE PIC X(1) VALUE "C".
50 77 MESS-TEXT      PIC X(254).
51 77 SQLSTATE-VAR   PIC X(5).
52 77 EX-COUNT       PIC S9(9) COMP-5.
53 77 COUNTER        PIC S9(9) VALUE 1 COMP-5.
54 EXEC SQL END DECLARE SECTION END-EXEC.
55 *
56 77 ZERO-FIELD     PIC X(5) VALUE "00000".
57 77 NOT-FND-FIELD  PIC X(5) VALUE "02000".
58 77 WHERE-ERROR    PIC X(72).
59 *
60 PROCEDURE DIVISION.
61 RESIDENT SECTION 1.
62 *
63 *Begin Main routine. Open the database, prepare a query,
64 *declare a cursor, open the cursor, fetch the cursor,
65 *and close the cursor.

```

```
66  *
67  MAIN.
68      DISPLAY ' '.
69      DISPLAY ' '.
70      DISPLAY 'DEMO2 SAMPLE ESQL PROGRAM RUNNING.'.
71      DISPLAY '  TEST OPEN USING AND FETCH INTO'.
72      DISPLAY ' '.
73      DISPLAY ' '.
74      DISPLAY 'PERFORM OPEN-DATABASE'.
75      DISPLAY ' '.
76      PERFORM OPEN-DATABASE.
77      DISPLAY ' '.
78      DISPLAY 'PERFORM PREPARE-QUERY'.
79      DISPLAY ' '.
80      PERFORM PREPARE-QUERY.
81      DISPLAY ' '.
82      DISPLAY 'PERFORM DECLARE-CURSOR'.
83      DISPLAY ' '.
84      PERFORM DECLARE-CURSOR.
85      DISPLAY ' '.
86      DISPLAY 'PERFORM OPEN-CURSOR'.
87      DISPLAY ' '.
88      PERFORM OPEN-CURSOR.
89      MOVE "00000" TO SQLSTATE-VAR.
90      DISPLAY ' '.
91      DISPLAY 'PERFORM FETCH-CURSOR'.
92      DISPLAY ' '.
93      PERFORM FETCH-CURSOR
94          UNTIL SQLSTATE-VAR NOT EQUAL TO ZERO-FIELD.
95      DISPLAY ' '.
96      DISPLAY 'PERFORM CLOSE-CURSOR'.
97      DISPLAY ' '.
98      PERFORM CLOSE-CURSOR.
99      EXEC SQL DISCONNECT CURRENT END-EXEC.
100     DISPLAY 'PROGRAM OVER'.
101  STOP RUN.
102  *
103  *Subroutine to open the database.
104  *
105  OPEN-DATABASE.
106      DISPLAY ' '.
107      DISPLAY 'EXECUTING CONNECT STATEMENT'.
108      DISPLAY ' '.
109      EXEC SQL CONNECT TO 'STORES7' END-EXEC.
110      IF SQLSTATE NOT EQUAL TO ZERO-FIELD
111          MOVE 'EXCEPTION ON DATABASE STORES7:'
112              TO WHERE-ERROR
113          PERFORM ERROR-PROCESS.
114  *
```



```

115 *Subroutine to prepare a query.
116 *
117   PREPARE-QUERY.

118       DISPLAY ' '.
119       DISPLAY 'EXECUTING PREPARE-QUERY STATEMENT'.
120       DISPLAY ' '.
121       EXEC SQL PREPARE QID FROM :DEMO-QUERY END-EXEC.
122       IF SQLSTATE NOT EQUAL TO ZERO-FIELD
123           MOVE 'ERROR ON PREPARE QUERY:'
124               TO WHERE-ERROR
125       PERFORM ERROR-PROCESS.

126 *
127 *Subroutine to declare a cursor.
128 *
129   DECLARE-CURSOR.
130       DISPLAY ' '.
131       DISPLAY 'EXECUTING DECLARE-CURSOR STATEMENT'.
132       DISPLAY ' '.
133       EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
134       IF SQLSTATE NOT EQUAL TO ZERO-FIELD
135           MOVE 'ERROR ON DECLARE CURSOR:'
136               TO WHERE-ERROR
137       PERFORM ERROR-PROCESS.

138 *
139 *Subroutine to open a cursor.
140 *
141   OPEN-CURSOR.
142       DISPLAY ' '.
143       DISPLAY 'EXECUTING OPEN-CURSOR STATEMENT'.
144       DISPLAY ' '.
145       EXEC SQL OPEN DEMOCURSOR USING :QUERY-VALUE END-EXEC.
146       IF SQLSTATE NOT EQUAL TO ZERO-FIELD
147           MOVE 'ERROR ON OPEN CURSOR:'
148               TO WHERE-ERROR
149       PERFORM ERROR-PROCESS.

150 *
151 *Subroutine to fetch a cursor. Display data (names).
152 *
153   FETCH-CURSOR.
154       DISPLAY ' '.
155       IF COUNTER IS EQUAL TO 1
156           DISPLAY 'EXECUTING FETCH-CURSOR STATEMENT'.
157       ADD 1 TO COUNTER.
158       DISPLAY ' '.
159       EXEC SQL FETCH DEMOCURSOR INTO :FNAME, :LNAME END-EXEC.
160       MOVE SQLSTATE TO SQLSTATE-VAR.
161       IF SQLSTATE NOT EQUAL TO ZERO-FIELD
162           AND

```

```
163         SQLSTATE NOT EQUAL TO NOT-FND-FIELD
164         MOVE 'ERROR DURING FETCH:'
165             TO WHERE-ERROR
166         PERFORM ERROR-PROCESS.
167     IF SQLSTATE IS EQUAL TO ZERO-FIELD
168         DISPLAY FNAME, ' ', LNAME.
169 *
170 *Subroutine to close a cursor.
171 *
172     CLOSE-CURSOR.
173         DISPLAY ' '.
174         DISPLAY 'EXECUTING CLOSE-CURSOR STATEMENT'.
175         DISPLAY ' '.
176         EXEC SQL CLOSE DEMOCURSOR END-EXEC.
177     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
178         MOVE 'ERROR ON OPEN CURSOR:'
179             TO WHERE-ERROR
180         PERFORM ERROR-PROCESS.
181 *
182 *Subroutine to check for exceptions.
183 *
184     ERROR-PROCESS.
185         DISPLAY WHERE-ERROR.
186         DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
187         DISPLAY '*****'.
188         EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
189         PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
190         IF SQLCODE NOT EQUAL TO ZERO
191             STOP RUN.
192 *
193 *Subroutine to print exception messages.
194 *
195     EX-LOOP.
196         EXEC SQL
197             GET DIAGNOSTICS EXCEPTION :COUNTER
198                 :MESS-TEXT=MESSAGE_TEXT
199         END-EXEC.
200         DISPLAY 'EXCEPTION ', COUNTER.
201         DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
202         DISPLAY '*****'.
203         ADD 1 TO COUNTER.
204 *
```

Explanation of DEMO2.ECO

The following subsections provide a paragraph-by-paragraph explanation of the DEMO2.ECO source program using representative COBOL sequence numbering to help you locate the line or concept under discussion.

In this example program, all paragraph headers begin in Area A. All SQL statements reside within Area B.

The [Informix Guide to SQL: Syntax](#) and Chapter 4, “Error Handling,” in this manual and describe the SQLSTATE value and the GET DIAGNOSTICS statement used in the DEMO2.ECO example program. The [Informix Guide to SQL: Syntax](#) provides a detailed description of the individual SQL statements used in DEMO2.ECO.

Lines 1 through 16

These lines of code define the standard COBOL IDENTIFICATION DIVISION that identify the program, and the ENVIRONMENT DIVISION, that identify the computer and specify any input/output devices that the program uses. This code segment uses the standard COBOL comment indicator, the asterisk (*) in position 7.

```

1 *
2 *This program, DEMO2, declares a cursor for a
3 *prepared statement with an unknown value. The
4 *value is supplied when the cursor is opened,
5 *the data is fetched, and the cursor is closed.
6 *Display the data (names).
7 *
8 IDENTIFICATION DIVISION.
9 PROGRAM-ID.
10 DEMO2.
11 *
12 ENVIRONMENT DIVISION.
13 CONFIGURATION SECTION.
14 SOURCE-COMPUTER. IFXSUN.
15 OBJECT-COMPUTER. IFXSUN.
16 *
```

Lines 17 through 36

The DATA DIVISION describes the files, records, and fields that the COBOL program uses.

You declare host variables, also known as independent data items, in the WORKING-STORAGE SECTION as level number 77. Here, the program defines the host variables FNAME, LNAME, DEMO-QUERY, and QUERY-VALUE as alphanumeric in the PICTURE clauses within the EXEC SQL BEGIN DECLARE SECTION END-EXEC and the EXEC SQL END DECLARE SECTION END-EXEC.

The host variables FNAME and LNAME represent columns in the **customer** table. The VALUE clause of the DEMO-QUERY data item represents an SQL statement that selects the first name and last name from the **customer** table, where the last name exceeds a specific value. The program uses a question mark wildcard in the WHERE clause of the SELECT statement and evaluates to a single character. The VALUE clause of the QUERY-VALUE data item establishes the initial value for the field as the letter "C".

The program defines the MESS-TEXT host variable to receive the contents of the MESSAGE-TEXT field in the GET DIAGNOSTICS statement. The program uses the EX-COUNT and COUNTER variables as conditions for exception-checking subroutines.

The program declares the non-host COBOL variables ZERO-FIELD and NOT-FND-FIELD to hold SQLSTATE values for **Success** ("00000") and **No Data Found** ("02000"), respectively. The program declares the non-host COBOL variable WHERE-ERROR as alphanumeric to hold an error message string.

```
17 *Declare variables.
18 *
19 DATA DIVISION.
20 WORKING-STORAGE SECTION.
21 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
22 77 FNAME PIC X(15).
23 77 LNAME PIC X(20).
24 77 DEMO-QUERY PIC X(50)
25 VALUE "SELECT FNAME, LNAME FROM CUSTOMER WHERE LNAME>?".
26 77 QUERY-VALUE PIC X(1) VALUE "C".
27 77 MESS-TEXT PIC X(254).
28 77 SQLSTATE-VAR PIC X(5).
29 77 EX-COUNT PIC S9(9) COMP-5.
30 77 COUNTER PIC S9(9) VALUE 1 COMP-5.
31 EXEC SQL END DECLARE SECTION END-EXEC.
32 *
33 77 ZERO-FIELD PIC X(5) VALUE "00000".
34 77 NOT-FND-FIELD PIC X(5) VALUE "02000".
35 77 WHERE-ERROR PIC X(72).
36 *
```

Lines 37 through 80

The MAIN paragraph begins in the PROCEDURE DIVISION. MAIN controls the sequence that performs each paragraph or subroutine and the iterations each paragraph or subroutine performs. The MAIN paragraph consists of instructions for the database server to perform the following steps. When each step executes successfully, the program performs those instructions in the following order:

1. Display messages on the screen to let you know that the ESQ/COBOL program DEMO2 runs successfully and the program currently tests the SQL statements and the OPEN DEMOCURSOR USING and FETCH DEMOCURSOR INTO clauses.
2. Open the database (in this case, **stores7**).
3. Prepare the query that was established in the WORKING-STORAGE section as DEMO-QUERY.
4. Declare the cursor DEMOCURSOR.
5. Open the cursor DEMOCURSOR.
6. Set the SQLSTATE-VAR variable to "00000".
7. Fetch the cursor DEMOCURSOR until the SQLSTATE-VAR variable does not equal to "00000".
8. Close the cursor DEMOCURSOR when the program meets that condition.
9. DISCONNECT from the current database.
10. Display "PROGRAM OVER" on the screen.
11. Execute the standard COBOL statement STOP RUN and exit the program, returning you to the operating system prompt.

```
37  *
38  PROCEDURE DIVISION.
39  RESIDENT SECTION 1.
40  *
41  *Begin Main routine.  Open the database, prepare a query,
42  *declare a cursor, open the cursor, fetch the cursor,
43  *and close the cursor.
44  *
45  MAIN.
46      DISPLAY ' '.
47      DISPLAY ' '.
48      DISPLAY 'DEMO2 SAMPLE ESQL PROGRAM RUNNING.'.
49      DISPLAY '  TEST OPEN USING AND FETCH INTO'.
50      DISPLAY ' '.
51      DISPLAY ' '.
52      DISPLAY 'PERFORM OPEN-DATABASE'.
53      DISPLAY ' '.
54      PERFORM OPEN-DATABASE.
55      DISPLAY ' '.
56      DISPLAY 'PERFORM PREPARE-QUERY'.
57      DISPLAY ' '.
58      PERFORM PREPARE-QUERY.
59      DISPLAY ' '.
60      DISPLAY 'PERFORM DECLARE-CURSOR'.
61      DISPLAY ' '.
62      PERFORM DECLARE-CURSOR.
63      DISPLAY ' '.
64      DISPLAY 'PERFORM OPEN-CURSOR'.
65      DISPLAY ' '.
66      PERFORM OPEN-CURSOR.
67      MOVE "00000" TO SQLSTATE-VAR.
68      DISPLAY ' '.
69      DISPLAY 'PERFORM FETCH-CURSOR'.
70      DISPLAY ' '.
71      PERFORM FETCH-CURSOR
72          UNTIL SQLSTATE-VAR NOT EQUAL TO ZERO-FIELD.
73      DISPLAY ' '.
74      DISPLAY 'PERFORM CLOSE-CURSOR'.
75      DISPLAY ' '.
76      PERFORM CLOSE-CURSOR.
77      EXEC SQL DISCONNECT CURRENT END-EXEC.
78      DISPLAY 'PROGRAM OVER'.
79  STOP RUN.
80  *
```

The following subsections individually describe the various PERFORM statements in the PROCEDURE division.

Lines 81 through 92

The OPEN-DATABASE subroutine opens the **stores7** database using the embedded SQL statement DATABASE. The SQL statement resides in the COBOL program between the words EXEC SQL and END-EXEC.

A conditional IF statement returns the message ERROR ON DATABASE STORES7 to the WHERE-ERROR variable when SQLSTATE does not equal "00000". Such an error can occur when you do not create the **stores7** database before you open it.

If an error occurs during the execution of the OPEN-DATABASE subroutine, the database server performs the ERROR-PROCESS subroutine.

```

81  *Subroutine to open the database.
82  *
83  OPEN-DATABASE.
84      DISPLAY ' '.
85      DISPLAY 'EXECUTING CONNECT STATEMENT'.
86      DISPLAY ' '.
87      EXEC SQL CONNECT TO 'STORES7' END-EXEC.
88      IF SQLSTATE NOT EQUAL TO ZERO-FIELD
89          MOVE 'EXCEPTION ON DATABASE STORES7:'
90              TO WHERE-ERROR
91          PERFORM ERROR-PROCESS.
92  *
```

Lines 93 through 104

The PREPARE-QUERY subroutine uses the embedded dynamic SQL statement PREPARE to prepare the QID query id variable from the DEMO-QUERY variable declared in the DATA DIVISION. When the program prepares the select cursor, the program passes the SELECT statement it represents to the database server.

An IF statement returns the message ERROR ON PREPARE QUERY to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the PREPARE-QUERY subroutine, the database server performs the ERROR-PROCESS subroutine.

```
93  *Subroutine to prepare a query.
94  *
95  PREPARE-QUERY.

96      DISPLAY ' '.
97      DISPLAY 'EXECUTING PREPARE-QUERY STATEMENT'.
98      DISPLAY ' '.
99      EXEC SQL PREPARE QID FROM :DEMO-QUERY END-EXEC.
100     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
101         MOVE 'ERROR ON PREPARE QUERY:'
102         TO WHERE-ERROR
103         PERFORM ERROR-PROCESS.
104  *
```

Lines 105 through 116

The DECLARE-CURSOR subroutine uses the embedded dynamic SQLstatement DECLARE to define DEMOCURSOR as a select cursor for the active set of rows that the prepared QID statement id variable specifies. DEMOCURSOR manages data that the program reads from the **customer** table.

The SELECT statement, that the embedded variable QID represents, determines the type of data that the program reads from the table. The QID variable represents a query on the first and last names of customers selected from the **customer** table, where the LNAME begins with a letter that has a higher ASCII value than the letter "C".

An IF statement returns the message ERROR ON DECLARE CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the DECLARE-CURSOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```

105 *Subroutine to declare a cursor.
106 *
107 DECLARE-CURSOR.
108     DISPLAY ' '.
109     DISPLAY 'EXECUTING DECLARE-CURSOR STATEMENT'.
110     DISPLAY ' '.
111     EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
112     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
113         MOVE 'ERROR ON DECLARE CURSOR:'
114         TO WHERE-ERROR
115     PERFORM ERROR-PROCESS.
116 *
```

Lines 117 through 128

The OPEN-CURSOR subroutine activates the select cursor DEMOCURSOR using the embedded dynamic SQL statement OPEN. The clause USING :QUERY-VALUE begins execution of the SELECT statement that the program variable represents.

When the select cursor was prepared, the SELECT statement it represents was passed to the database server. Here, the program passes the values specified in the USING clause to the database server. Rather than construct the first row of the active set for the query, the database server sets the SQLSTATE value. When the program uses a valid SELECT, the value of SQLSTATE equals "00000".

An IF statement returns the message ERROR ON OPEN CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the OPEN-CURSOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```
117 *Subroutine to open a cursor.
118 *
119 OPEN-CURSOR.
120     DISPLAY ' '.
121     DISPLAY 'EXECUTING OPEN-CURSOR STATEMENT'.
122     DISPLAY ' '.
123     EXEC SQL OPEN DEMOCURSOR USING :QUERY-VALUE END-EXEC.
124     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
125         MOVE 'ERROR ON OPEN CURSOR:'
126         TO WHERE-ERROR
127     PERFORM ERROR-PROCESS.
128 *
```

Lines 129 through 147

The FETCH-CURSOR subroutine uses the embedded dynamic SQL statement FETCH to move the previously declared cursor DEMOCURSOR to a new row in the active set and to retrieve the row values into memory for the COBOL program to use. DEMOCURSOR selects a row from the **customer** table and puts the data from that row into the host variables FNAME and LNAME.

The PERFORM-UNTIL statement in the MAIN routine uses the SQLSTATE-VAR variable to check the result of the FETCH statement. As long as SQLSTATE equals "00000", the data was fetched successfully, and the subroutine continues. When the program has fetched all the data that meets the criteria, the database server sets SQLSTATE to "02000" to indicate that no more data exists. It also displays the first and last names of the customers on the screen.

An IF statement returns the message `ERROR DURING FETCH` to the `WHERE-ERROR` variable when `SQLSTATE` does not equal "00000" *and* when `SQLSTATE` does not equal "02000". When an error occurs during the execution of the `FETCH-CURSOR` subroutine, the database server performs the `ERROR-PROCESS` subroutine.

```

129 *Subroutine to fetch a cursor. Display data (names).
130 *
131  FETCH-CURSOR.
132      DISPLAY ' '.
133      IF COUNTER IS EQUAL TO 1
134          DISPLAY 'EXECUTING FETCH-CURSOR STATEMENT'.
135      ADD 1 TO COUNTER.
136      DISPLAY ' '.
137      EXEC SQL FETCH DEMOCURSOR INTO :FNAME, :LNAME END-EXEC.
138      MOVE SQLSTATE TO SQLSTATE-VAR.
139      IF SQLSTATE NOT EQUAL TO ZERO-FIELD
140          AND
141          SQLSTATE NOT EQUAL TO NOT-FND-FIELD
142          MOVE 'ERROR DURING FETCH:'
143              TO WHERE-ERROR
144          PERFORM ERROR-PROCESS.
145      IF SQLSTATE IS EQUAL TO ZERO-FIELD
146          DISPLAY FNAME, ' ', LNAME.
147  *
```

Lines 148 through 159

The `CLOSE-CURSOR` subroutine closes the cursor `DEMOCURSOR` using the embedded dynamic SQL statement `CLOSE`. It disassociates the cursor from the `SELECT` statement and stops the query process.

An IF statement returns the message `ERROR ON OPEN CURSOR` to the `WHERE-ERROR` variable when `SQLSTATE` does not equal "00000". When an error occurs during the execution of the `CLOSE-CURSOR` subroutine, the database server performs the `ERROR-PROCESS` subroutine.

```

148 *Subroutine to close a cursor.
149 *
150  CLOSE-CURSOR.
151      DISPLAY ' '.
152      DISPLAY 'EXECUTING CLOSE-CURSOR STATEMENT'.
153      DISPLAY ' '.
154      EXEC SQL CLOSE DEMOCURSOR END-EXEC.
155      IF SQLSTATE NOT EQUAL TO ZERO-FIELD
156          MOVE 'ERROR ON OPEN CURSOR:'
```

```
157             TO WHERE-ERROR
158     PERFORM ERROR-PROCESS.
159 *
```

Lines 160 through 170

The ERROR-PROCESS subroutine contains the process that counts SQLSTATE exceptions. That subroutine executes whenever an error occurs in one of the other subroutines. The DEMO2.ECO program stops running whenever the ERROR-PROCESS subroutine receives an SQLCODE value not equal to ZERO.

SQLSTATE indicates the result of executing an SQL statement ("00000", "02000", or a value greater than "02000"). The GET DIAGNOSTICS NUMBER field contains the count of exceptions associated with the SQLSTATE code. The PERFORM UNTIL statement executes the EX-LOOP subroutine that displays an error message for each exception.

```
160 *Subroutine to check for exceptions.
161 *
162     ERROR-PROCESS.
163         DISPLAY WHERE-ERROR.
164         DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
165         DISPLAY '*****'.
166         EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
167         PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
168         IF SQLCODE NOT EQUAL TO ZERO
169             STOP RUN.
170 *
```

Lines 171 through 180

The EX-LOOP subroutine displays the exception number and the error message for each SQLSTATE exception.

```
171     EX-LOOP.
172         EXEC SQL
173             GET DIAGNOSTICS EXCEPTION :COUNTER
174             :MESS-TEXT=MESSAGE_TEXT
175         END-EXEC.
176         DISPLAY 'EXCEPTION ', COUNTER.
177         DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
178         DISPLAY '*****'.
179         ADD 1 TO COUNTER.
180 *
```

The DEMO3.ECO Program

DEMO3.ECO uses system descriptors with a SELECT statement that receives WHERE-clause values at run time.

```

1 *
2 *This program, DEMO3, uses system
3 *descriptors with a SELECT statement that receives
4 *WHERE-clause values at run time. DEMO3 uses a
5 *PREPARE statement on a query with an unknown
6 *parameter in the WHERE clause and includes the
7 *DESCRIBE, GET DESCRIPTOR, ALLOCATE DESCRIPTOR,
8 *and DEALLOCATE DESCRIPTOR statements. DEMO3
9 *also prints the contents of the system descriptor
10 *area.
11 *
12 IDENTIFICATION DIVISION.
13 PROGRAM-ID.
14     DEMO3.
15 *
16 ENVIRONMENT DIVISION.
17 CONFIGURATION SECTION.
18 SOURCE-COMPUTER. IFXSUN.
19 OBJECT-COMPUTER. IFXSUN.
20 *
21 *Declare variables.
22 *
23 DATA DIVISION.
24 WORKING-STORAGE SECTION.
25 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
26 77 RET-BUFFER PIC X(20).
27 77 DEMO-QUERY PIC X(50)
28     VALUE "SELECT FNAME, LNAME FROM CUSTOMER WHERE LNAME>?".
29 77 QUERY-VALUE PIC X(1) VALUE "C".
30 77 DESC-COUNT SQLINT.
31 77 DESC-INDEX SQLINT.
32 77 MESS-TEXT PIC X(254).
33 77 SQLSTATE-VAR PIC X(5).
34 77 EX-COUNT PIC S9(9) COMP-5.
35 77 COUNTER PIC S9(9) VALUE 1 COMP-5.
36 EXEC SQL END DECLARE SECTION END-EXEC.
37 *
38 77 ZERO-FIELD PIC X(5) VALUE "00000".
39 77 NOT-FND-FIELD PIC X(5) VALUE "02000".
40 77 WHERE-ERROR PIC X(72).
41 77 COUNTER-F PIC S9(9) VALUE 1 COMP-5.
42 77 COUNT-DESC PIC S9(9) VALUE 1 COMP-5.

```

```
43  *
44  PROCEDURE DIVISION.
45  RESIDENT SECTION 1.
46  *
47  *Begin Main routine. Open a database, prepare a query,
48  *declare a cursor, allocate descriptors, execute a
49  *describe statement, open the cursor, fetch the cursor,
50  *and close the cursor.
51  *
52  MAIN.
53      DISPLAY ' '.
54      DISPLAY ' '.
55      DISPLAY 'DEMO3 SAMPLE ESQL PROGRAM RUNNING.'.
56      DISPLAY '  TEST SIMPLE DECLARE/OPEN/FETCH/LOOP'.
57      DISPLAY ' '.
58      DISPLAY ' '.
59      DISPLAY 'PERFORM OPEN-DATABASE'.
60      DISPLAY ' '.
61      PERFORM OPEN-DATABASE.
62      DISPLAY ' '.
63      DISPLAY 'PERFORM PREPARE-QUERY'.
64      DISPLAY ' '.
65      PERFORM PREPARE-QUERY.
66      DISPLAY ' '.
67      DISPLAY 'PERFORM DECLARE-CURSOR'.
68      DISPLAY ' '.
69      PERFORM DECLARE-CURSOR.
70      DISPLAY ' '.
71      DISPLAY 'PERFORM ALLOCATE-DESCRIPTOR'.
72      DISPLAY ' '.
73      PERFORM ALLOCATE-DESCRIPTOR.
74      DISPLAY ' '.
75      DISPLAY 'PERFORM DESCRIBE-ID'.
76      DISPLAY ' '.
77      PERFORM DESCRIBE-ID.
78      DISPLAY ' '.
79      DISPLAY 'PERFORM OPEN-CURSOR'.
80      DISPLAY ' '.
81      PERFORM OPEN-CURSOR.
82      MOVE "00000" TO SQLSTATE-VAR.
83      DISPLAY ' '.
84      DISPLAY 'PERFORM FETCH-CURSOR'.
85      DISPLAY ' '.
86      PERFORM FETCH-CURSOR
87          UNTIL SQLSTATE-VAR NOT EQUAL TO ZERO-FIELD.
88      DISPLAY ' '.
89      DISPLAY 'PERFORM CLOSE-CURSOR'.
90      DISPLAY ' '.
91      PERFORM CLOSE-CURSOR.
```

```

92      EXEC SQL DISCONNECT CURRENT END-EXEC.
93      DISPLAY 'PROGRAM OVER'.
94  STOP RUN.
95  *
96  *Subroutine to open a database.
97  *
98  OPEN-DATABASE.
99      DISPLAY ' '.
100     DISPLAY 'EXECUTING CONNECT STATEMENT'.
101     DISPLAY ' '.
102     EXEC SQL CONNECT TO 'STORES7' END-EXEC.
103     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
104         MOVE 'EXCEPTION ON DATABASE STORES7:'
105         TO WHERE-ERROR
106         PERFORM ERROR-PROCESS.
107  *
108  *Subroutine to prepare a query.
109  *
110  PREPARE-QUERY.
111     DISPLAY ' '.
112     DISPLAY 'EXECUTING PREPARE STATEMENT'.
113     DISPLAY ' '.
114     EXEC SQL PREPARE QID FROM :DEMO-QUERY END-EXEC.
115     IF SQLSTATE IS NOT EQUAL TO ZERO-FIELD
116         MOVE 'ERROR ON PREPARE QUERY:'
117         TO WHERE-ERROR
118         PERFORM ERROR-PROCESS.
119  *
120  *Subroutine to declare a cursor.
121  *
122  DECLARE-CURSOR.
123     DISPLAY ' '.
124     DISPLAY 'EXECUTING DECLARE-CURSOR STATEMENT'.
125     DISPLAY ' '.
126     EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
127     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
128         MOVE 'ERROR ON DECLARE CURSOR:'
129         TO WHERE-ERROR
130         PERFORM ERROR-PROCESS.
131  *
132  *Subroutine to allocate a descriptor.
133  *
134  ALLOCATE-DESCRIPTOR.
135     DISPLAY ' '.
136     DISPLAY 'EXECUTING ALLOCATE DESCRIPTOR STATEMENT'.
137     DISPLAY ' '.
138     EXEC SQL ALLOCATE DESCRIPTOR 'desc' END-EXEC.
139     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
140         MOVE 'ERROR ON ALLOCATE DESCRIPTOR:'

```

```
141         TO WHERE-ERROR
142     PERFORM ERROR-PROCESS.
143 *
144 *Subroutine to deallocate a descriptor.
145 *
146     DEALLOCATE-DESCRIPTOR.
147     DISPLAY ' '.
148     DISPLAY 'EXECUTING DEALLOCATE DESCRIPTOR STATEMENT'.
149     DISPLAY ' '.
150     EXEC SQL DEALLOCATE DESCRIPTOR 'desc' END-EXEC.
151     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
152         MOVE 'ERROR ON DEALLOCATE DESCRIPTOR:'
153         TO WHERE-ERROR
154     PERFORM ERROR-PROCESS.
155 *
156 *Subroutine to execute a describe statement.
157 *
158     DESCRIBE-ID.
159     DISPLAY ' '.
160     DISPLAY 'EXECUTING DESCRIBE STATEMENT'.
161     DISPLAY ' '.
162     EXEC SQL DESCRIBE QID
163         USING SQL DESCRIPTOR 'desc' END-EXEC.
164     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
165         MOVE 'ERROR ON DESCRIBE ID:'
166         TO WHERE-ERROR
167     PERFORM ERROR-PROCESS.
168     EXEC SQL GET DESCRIPTOR 'desc'
169         :DESC-COUNT=COUNT END-EXEC.
170     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
171         MOVE 'ERROR ON GET DESCRIPTOR:'
172         TO WHERE-ERROR
173     PERFORM ERROR-PROCESS.
174 *
175 *Subroutine to open a cursor.
176 *
177     OPEN-CURSOR.
178     DISPLAY ' '.
179     DISPLAY 'EXECUTING OPEN-CURSOR STATEMENT'.
180     DISPLAY ' '.
181     EXEC SQL OPEN DEMOCURSOR USING :QUERY-VALUE END-EXEC.
182     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
183         MOVE 'ERROR ON OPEN CURSOR:'
184         TO WHERE-ERROR
185     PERFORM ERROR-PROCESS.
186 *
187 *Subroutine to fetch a cursor.
188 *
189     FETCH-CURSOR.
```



```

190     DISPLAY ' '.
191     IF COUNTER-F IS EQUAL TO 1
192     DISPLAY 'EXECUTING FETCH-CURSOR STATEMENT'.
193     ADD 1 TO COUNTER-F.
194     DISPLAY ' '.
195     EXEC SQL FETCH DEMOCURSOR
196     USING SQL DESCRIPTOR 'desc' END-EXEC.
197     MOVE SQLSTATE TO SQLSTATE-VAR.
198     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
199         AND
200         SQLSTATE NOT EQUAL TO NOT-FND-FIELD
201         MOVE 'ERROR DURING FETCH:'
202         TO WHERE-ERROR
203         PERFORM ERROR-PROCESS.
204     IF SQLSTATE NOT EQUAL TO NOT-FND-FIELD
205         PERFORM PRINT-DESCRIPTOR
206         VARYING DESC-INDEX FROM 1 BY 1
207         UNTIL DESC-INDEX>DESC-COUNT
208     DISPLAY ' '.
209 *
210 *Subroutine to print a descriptor. Display data (names).
211 *
212     PRINT-DESCRIPTOR.
213     DISPLAY ' '.
214     IF COUNT-DESC IS EQUAL TO 1
215     DISPLAY 'EXECUTING GET DESCRIPTOR STATEMENTS'.
216     ADD 1 TO COUNT-DESC.
217     DISPLAY ' '.
218     EXEC SQL GET DESCRIPTOR 'desc' VALUE :DESC-INDEX
219     :RET-BUFFER=DATA END-EXEC.
220     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
221     MOVE 'ERROR ON GET DESCRIPTOR:'
222     TO WHERE-ERROR
223     PERFORM ERROR-PROCESS.
224     DISPLAY RET-BUFFER, ' ' WITH NO ADVANCING.
225 *
226 *Subroutine to close a cursor.
227 *
228     CLOSE-CURSOR.
229     DISPLAY ' '.
230     DISPLAY 'EXECUTING CLOSE-CURSOR STATEMENT'.
231     DISPLAY ' '.
232     EXEC SQL CLOSE DEMOCURSOR END-EXEC.
233     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
234     MOVE 'ERROR ON OPEN CURSOR:'
235     TO WHERE-ERROR
236     PERFORM ERROR-PROCESS.
237 *
238 *Subroutine to check for exceptions.

```

```
239 *
240 ERROR-PROCESS.
241     DISPLAY WHERE-ERROR.
242     DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
243     DISPLAY '*****'.
244     EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
245     PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
246     IF SQLCODE NOT EQUAL TO ZERO
247         STOP RUN.
248 *
249 *Subroutine to print exception messages.
250 *
251 EX-LOOP.
252     EXEC SQL
253         GET DIAGNOSTICS EXCEPTION :COUNTER
254         :MESS-TEXT=MESSAGE_TEXT
255     END-EXEC.
256     DISPLAY 'EXCEPTION ', COUNTER.
257     DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
258     DISPLAY '*****'.
259     ADD 1 TO COUNTER.
260 *
```

Explanation of DEMO3.ECO

The following subsections provide a paragraph-by-paragraph explanation of the DEMO3.ECO source program using representative COBOL sequence numbering to help you locate the line or concept under discussion.

In this example program, all paragraph headers begin in Area A. All SQL statements reside within Area B.

Chapter 4, “Error Handling,” of this manual and the *Informix Guide to SQL: Syntax* describe the SQLSTATE value and the GET DIAGNOSTICS statement that the DEMO3.ECO program uses. The *Informix Guide to SQL: Syntax* provides a detailed description of the individual SQL statements used in DEMO2.ECO.

Lines 1 through 20

These lines of code define the standard COBOL IDENTIFICATION DIVISION and ENVIRONMENT DIVISION used to identify the program, the computer, and any input/output devices.

```

1 *
2 *This program, DEMO3, uses system
3 *descriptors with a SELECT statement that receives
4 *WHERE-clause values at run time. DEMO3 uses a
5 *PREPARE statement on a query with an unknown
6 *parameter in the WHERE clause and includes the
7 *DESCRIBE, GET DESCRIPTOR, ALLOCATE DESCRIPTOR,
8 *and DEALLOCATE DESCRIPTOR statements. DEMO3
9 *also prints the contents of the system descriptor
10 *area.
11 *
12 IDENTIFICATION DIVISION.
13 PROGRAM-ID.
14     DEMO3.
15 *
16 ENVIRONMENT DIVISION.
17 CONFIGURATION SECTION.
18 SOURCE-COMPUTER. IFXSUN.
19 OBJECT-COMPUTER. IFXSUN.
20 *
```

Lines 21 through 43

The DATA DIVISION describes the files, records, and fields that the COBOL program uses.

You declare host variables, also known as independent data items, in the WORKING-STORAGE SECTION as level number 77. Here, the program defines host variables RET-BUFFER, DEMO-QUERY, QUERY-VALUE, DESC-COUNT, and DESC-INDEX in the PICTURE clauses within the EXEC SQL BEGIN DECLARE SECTION END-EXEC and the EXEC SQL END DECLARE SECTION END-EXEC.

- The program declares RET-BUFFER, DEMO-QUERY, and QUERY-VALUE as alphanumeric. The program designates RET-BUFFER as the buffer that holds the results of the query DEMO-QUERY.
 - The VALUE clause of the DEMO-QUERY data item represents an SQL statement that selects the first name and last name from the **customer** table where the last name exceeds a specific value. The program uses a question mark (?) wildcard in the WHERE clause of the SELECT statement and evaluates to a single character.
 - The VALUE clause of the QUERY-VALUE data item establishes the initial value for the field as the letter "C".
- The program declares DESC-COUNT and DESC-INDEX as the predefined data type SQLINT. DESC-COUNT holds a descriptor value and DESC-INDEX serves as a counter.

The program defines the MESS-TEXT host variable to receive the contents of the MESSAGE-TEXT field in the GET DIAGNOSTICS statement. The program uses the EX-COUNT and COUNTER variables as conditions for exception-checking subroutines.

The program declares non-host COBOL variables ZERO-FIELD and NOT-FND-FIELD to hold SQLSTATE values for **Success** ("00000") and **No Data Found** ("02000"), respectively. The program declares the non-host COBOL variable WHERE-ERROR as alphanumeric to hold an error message string. The program also declares non-host COBOL variables COUNTER-F and COUNT-DESC to print a statement once at the beginning of a fetch subroutine and at the beginning of a print-descriptor subroutine, respectively.

```

21 *Declare variables.
22 *
23 DATA DIVISION.
24 WORKING-STORAGE SECTION.
25 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
26 77 RET-BUFFER PIC X(20).
27 77 DEMO-QUERY PIC X(50)
28    VALUE "SELECT FNAME, LNAME FROM CUSTOMER WHERE LNAME>?".
29 77 QUERY-VALUE PIC X(1) VALUE "C".
30 77 DESC-COUNT SQLINT.
31 77 DESC-INDEX SQLINT.
32 77 MESS-TEXT PIC X(254).
33 77 SQLSTATE-VAR PIC X(5).
34 77 EX-COUNT PIC S9(9) COMP-5.
35 77 COUNTER PIC S9(9) VALUE 1 COMP-5.
36 EXEC SQL END DECLARE SECTION END-EXEC.
37 *
38 77 ZERO-FIELD PIC X(5) VALUE "00000".
39 77 NOT-FND-FIELD PIC X(5) VALUE "02000".
40 77 WHERE-ERROR PIC X(72).
41 77 COUNTER-F PIC S9(9) VALUE 1 COMP-5.
42 77 COUNT-DESC PIC S9(9) VALUE 1 COMP-5.
43 *
```

Lines 44 through 95

The MAIN paragraph begins in the PROCEDURE DIVISION. MAIN controls the sequence that performs each paragraph or subroutine and the iterations for each paragraph or subroutine. The MAIN paragraph consists of instructions for the database server to perform certain steps. When each step executes successfully, the program performs those instructions in the following order:

1. Display messages on the screen to let you know that the ESQL/COBOL program DEMO3 runs successfully and that the program tests the SQL statements and clauses OPEN USING SQL DESCRIPTOR and FETCH USING SQL DESCRIPTOR.
2. Open the database (in this case, **stores7**).

3. Prepare the query that was established in the WORKING-STORAGE section as DEMO-QUERY.
4. Declare the cursor DEMOCURSOR.
5. Allocate a descriptor.
6. Obtain information about the prepared SELECT statement.
7. Open the cursor DEMOCURSOR.
8. Set SQLSTATE-VAR variable to "00000" and fetch the cursor DEMOCURSOR until SQLSTATE-VAR does not equal "00000".
9. Close the cursor DEMOCURSOR when the program meets that condition.
10. Deallocate the previously allocated descriptor.
11. Disconnect from the current database.
12. Display "PROGRAM OVER" on the screen.
13. Execute the standard COBOL statement STOP RUN and exit the program, returning you to the operating system prompt.

```

44  PROCEDURE DIVISION.
45  RESIDENT SECTION 1.
46  *
47  *Begin Main routine. Open a database, prepare a query.
48  *declare a cursor, allocate descriptors, execute a
49  *describe statement, open the cursor, fetch the cursor,
50  *and close the cursor.
51  *
52  MAIN.
53      DISPLAY ' '.
54      DISPLAY ' '.
55      DISPLAY 'DEMO3 SAMPLE ESQL PROGRAM RUNNING.'.
56      DISPLAY '  TEST SIMPLE DECLARE/OPEN/FETCH/LOOP'.
57      DISPLAY ' '.
58      DISPLAY ' '.
59      DISPLAY 'PERFORM OPEN-DATABASE'.
60      DISPLAY ' '.
61      PERFORM OPEN-DATABASE.
62      DISPLAY ' '.
63      DISPLAY 'PERFORM PREPARE-QUERY'.
64      DISPLAY ' '.
65      PERFORM PREPARE-QUERY.
66      DISPLAY ' '.
67      DISPLAY 'PERFORM DECLARE-CURSOR'.
68      DISPLAY ' '.
69      PERFORM DECLARE-CURSOR.
70      DISPLAY ' '.
71      DISPLAY 'PERFORM ALLOCATE-DESCRIPTOR'.
72      DISPLAY ' '.
73      PERFORM ALLOCATE-DESCRIPTOR.
74      DISPLAY ' '.
75      DISPLAY 'PERFORM DESCRIBE-ID'.
76      DISPLAY ' '.
77      PERFORM DESCRIBE-ID.
78      DISPLAY ' '.
79      DISPLAY 'PERFORM OPEN-CURSOR'.
80      DISPLAY ' '.
81      PERFORM OPEN-CURSOR.
82      MOVE "00000" TO SQLSTATE-VAR.
83      DISPLAY ' '.
84      DISPLAY 'PERFORM FETCH-CURSOR'.
85      DISPLAY ' '.
86      PERFORM FETCH-CURSOR
87          UNTIL SQLSTATE-VAR NOT EQUAL TO ZERO-FIELD.
88      DISPLAY ' '.
89      DISPLAY 'PERFORM CLOSE-CURSOR'.
90      DISPLAY ' '.
91      PERFORM CLOSE-CURSOR.
92      EXEC SQL DISCONNECT CURRENT END-EXEC.

```

```
93      DISPLAY 'PROGRAM OVER'.
94      STOP RUN.
95      *
```

The following subsections individually describe the various PERFORM statements that appear in the PROCEDURE division.

Lines 96 through 107

The OPEN-DATABASE subroutine opens the **stores7** database using the CONNECT embedded SQL statement. The SQL statement resides in the COBOL program between the words EXEC SQL and END-EXEC.

A conditional IF statement returns the message ERROR ON DATABASE STORES7 to the WHERE-ERROR variable when SQLSTATE does not equal "00000". Such an error can occur when you do not create the **stores7** database before you open it.

If an error occurs during the execution of the OPEN-DATABASE subroutine, the database server performs the ERROR-PROCESS subroutine.

```
96      *Subroutine to open a database.
97      *
98      OPEN-DATABASE.
99          DISPLAY ' '.
100         DISPLAY 'EXECUTING CONNECT STATEMENT'.
101         DISPLAY ' '.
102         EXEC SQL CONNECT TO 'STORES7' END-EXEC.
103         IF SQLSTATE NOT EQUAL TO ZERO-FIELD
104             MOVE 'EXCEPTION ON DATABASE STORES7:'
105             TO WHERE-ERROR
106             PERFORM ERROR-PROCESS.
107      *
```

Lines 108 through 119

The PREPARE-QUERY subroutine uses the embedded dynamic SQL statement PREPARE to prepare the query id variable QID from the DEMO-QUERY variable declared in the DATA DIVISION. When the program prepares the select cursor, the program passes the SELECT statement it represents to the database server for a query on the **stores7** database.

An IF statement returns the message ERROR ON PREPARE QUERY to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the PREPARE-QUERY subroutine, the database server performs the ERROR-PROCESS subroutine.

```

108 *Subroutine to prepare a query.
109 *
110 PREPARE-QUERY.
111     DISPLAY ' '.
112     DISPLAY 'EXECUTING PREPARE STATEMENT'.
113     DISPLAY ' '.
114     EXEC SQL PREPARE QID FROM :DEMO-QUERY END-EXEC.
115     IF SQLSTATE IS NOT EQUAL TO ZERO-FIELD
116         MOVE 'ERROR ON PREPARE QUERY:'
117         TO WHERE-ERROR
118     PERFORM ERROR-PROCESS.
119 *
```

Lines 120 through 131

The DECLARE-CURSOR subroutine uses the embedded dynamic SQL statement DECLARE to define DEMOCURSOR as a select cursor for the active set of rows that the prepared QID statement id variable specifies. The program uses DEMOCURSOR to manage data that the program reads from the **customer** table.

The SELECT statement, that the embedded variable QID represents, determines the type of data that the program reads from the table. The QID variable represents a query on the first and last names of customers selected from the **customer** table, where the LNAME begins with a letter that has a higher ASCII value than the letter "C".

An IF statement returns the message `ERROR ON DECLARE CURSOR` to the `WHERE-ERROR` variable when `SQLSTATE` does not equal "00000". When an error occurs during the execution of the `DECLARE-CURSOR` subroutine, the database server performs the `ERROR-PROCESS` subroutine.

```
120 *Subroutine to declare a cursor.
121 *
122 DECLARE-CURSOR.
123     DISPLAY ' '.
124     DISPLAY 'EXECUTING DECLARE-CURSOR STATEMENT'.
125     DISPLAY ' '.
126     EXEC SQL DECLARE DEMOCURSOR CURSOR FOR QID END-EXEC.
127     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
128         MOVE 'ERROR ON DECLARE CURSOR:'
129             TO WHERE-ERROR
130     PERFORM ERROR-PROCESS.
131 *
```

Lines 132 through 143

The `ALLOCATE-DESCRIPTOR` subroutine uses the embedded dynamic SQL statement `ALLOCATE DESCRIPTOR` to allocate space in memory for a system descriptor area that the quoted string `desc` identifies.

An IF statement returns the message `ERROR ON ALLOCATE DESCRIPTOR` to the `WHERE-ERROR` variable when `SQLSTATE` does not equal "00000". When an error occurs during the execution of the `ALLOCATE-DESCRIPTOR` subroutine, the database server performs the `ERROR-PROCESS` subroutine.

```
132 *Subroutine to allocate a descriptor.
133 *
134 ALLOCATE-DESCRIPTOR.
135     DISPLAY ' '.
136     DISPLAY 'EXECUTING ALLOCATE DESCRIPTOR STATEMENT'.
137     DISPLAY ' '.
138     EXEC SQL ALLOCATE DESCRIPTOR 'desc' END-EXEC.
139     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
140         MOVE 'ERROR ON ALLOCATE DESCRIPTOR:'
141             TO WHERE-ERROR
142     PERFORM ERROR-PROCESS.
143 *
```

Lines 144 through 155

The DEALLOCATE-DESCRIPTOR subroutine uses the embedded dynamic SQL statement DEALLOCATE DESCRIPTOR to free space previously allocated in memory for a system descriptor area that the quoted string **desc** identifies. When the program deallocates a system descriptor, the program frees all associated item descriptors and memory for data values.

An IF statement returns the message ERROR ON DEALLOCATE DESCRIPTOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the DEALLOCATE-DESCRIPTOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```

144 *Subroutine to deallocate a descriptor.
145 *
146 DEALLOCATE-DESCRIPTOR.
147     DISPLAY ' '.
148     DISPLAY 'EXECUTING DEALLOCATE DESCRIPTOR STATEMENT'.
149     DISPLAY ' '.
150     EXEC SQL DEALLOCATE DESCRIPTOR 'desc' END-EXEC.
151     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
152         MOVE 'ERROR ON DEALLOCATE DESCRIPTOR:'
153             TO WHERE-ERROR
154     PERFORM ERROR-PROCESS.
155 *
```

Lines 156 through 174

The DESCRIBE-ID subroutine uses the embedded dynamic SQL statements DESCRIBE and GET DESCRIPTOR to obtain information about the previously allocated system descriptor area that the quoted string **desc** identifies.

The DESCRIBE statement returns the statement type and the number, data types, and size of the values returned using the query on the LNAME and FNAME columns in the **customer** table. QID is the data structure that represents the prepared SELECT statement.

The GET DESCRIPTOR statement gets values from the system descriptor area that the descriptor **desc** identifies. The program sets the host variable :DESC-COUNT to a value that represents the number of columns, in this case, 2 (for FNAME and LNAME).

An IF statement returns the message ERROR ON DESCRIBE to the screen if SQLSTATE does not equal "00000" when the DESCRIBE statement is executed. Another IF statement returns the message ERROR ON GET DESCRIPTOR to the screen when SQLSTATE does not equal "00000" when that statement executes. When an error occurs during the execution of the DESCRIBE-ID subroutine, the database server performs the ERROR-PROCESS subroutine.

```
156 *Subroutine to execute a describe statement.
157 *
158 DESCRIBE-ID.
159     DISPLAY ' '.
160     DISPLAY 'EXECUTING DESCRIBE STATEMENT'.
161     DISPLAY ' '.
162     EXEC SQL DESCRIBE QID
163         USING SQL DESCRIPTOR 'desc' END-EXEC.
164     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
165         MOVE 'ERROR ON DESCRIBE ID:'
166         TO WHERE-ERROR
167     PERFORM ERROR-PROCESS.
168     EXEC SQL GET DESCRIPTOR 'desc'
169         :DESC-COUNT=COUNT END-EXEC.
170     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
171         MOVE 'ERROR ON GET DESCRIPTOR:'
172         TO WHERE-ERROR
173     PERFORM ERROR-PROCESS.
174 *
```

Lines 175 through 186

The OPEN-CURSOR subroutine activates the select cursor DEMOCURSOR using the embedded dynamic SQL statement OPEN. The clause USING :QUERY-VALUE begins execution of the SELECT statement that the program variable represents.

When the select cursor was prepared, the SELECT statement it represents was passed to the database server. Here, the program also passes values specified in the USING clause to the database server. Rather than construct the first row of the active set for the query, the database server sets the SQLSTATE value. When the program uses a valid SELECT statement, the program sets the value of SQLSTATE to "00000".

An IF statement returns the message ERROR ON OPEN CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the OPEN-CURSOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```

175 *Subroutine to open a cursor.
176 *
177 OPEN-CURSOR.
178     DISPLAY ' '.
179     DISPLAY 'EXECUTING OPEN-CURSOR STATEMENT'.
180     DISPLAY ' '.
181     EXEC SQL OPEN DEMOCURSOR USING :QUERY-VALUE END-EXEC.
182     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
183         MOVE 'ERROR ON OPEN CURSOR:'
184         TO WHERE-ERROR
185     PERFORM ERROR-PROCESS.
186 *
```

Lines 187 through 209

The FETCH-CURSOR subroutine uses the embedded dynamic SQL statement FETCH to move the previously declared cursor DEMOCURSOR to a new row in the active set and to retrieve the row values into memory for the COBOL program to use. The FETCH statement uses the SQL descriptor **desc** that was previously allocated in the ALLOCATE-DESCRIPTOR subroutine.

The PERFORM-UNTIL statement in the MAIN routine uses the SQLSTATE-VAR variable **i** to check the result of the FETCH statement. As long as SQLSTATE equals "00000", the program successfully fetched the data, and the subroutine continues. When the program fetches all the data that meets the criteria, the database server sets SQLSTATE to "02000" to indicate that no data exists.

As long as SQLSTATE does not equal "02000", meaning that not all of the data has been fetched yet, the database server performs the PRINT-DESCRIPTOR subroutine. The DEMO3.ECO program increments the DESC-INDEX descriptor counter (initialized as 1) by 1 each time the program fetches a column in the row, until the program meets the condition where the DESC-INDEX value exceeds the DESC-COUNT value (2). The subroutine then displays the row (FNAME and LNAME) on the screen. The conditional PERFORM-UNTIL statement tests the condition before the program executes the set of instructions.

An IF statement returns the message `ERROR DURING FETCH` to the `WHERE-ERROR` variable when `SQLSTATE` does not equal "00000" *and* when `SQLSTATE` does not equal "02000". When an error occurs during the execution of the `FETCH-CURSOR` subroutine, the database server performs the `ERROR-PROCESS` subroutine instead of the `PRINT-DESCRIPTOR` subroutine.

```
187 *Subroutine to fetch a cursor.
188 *
189  FETCH-CURSOR.
190      DISPLAY ' '.
191      IF COUNTER-F IS EQUAL TO 1
192          DISPLAY 'EXECUTING FETCH-CURSOR STATEMENT'.
193          ADD 1 TO COUNTER-F.
194          DISPLAY ' '.
195          EXEC SQL FETCH DEMOCURSOR
196              USING SQL DESCRIPTOR 'desc' END-EXEC.
197          MOVE SQLSTATE TO SQLSTATE-VAR.
198          IF SQLSTATE NOT EQUAL TO ZERO-FIELD
199              AND
200              SQLSTATE NOT EQUAL TO NOT-FND-FIELD
201              MOVE 'ERROR DURING FETCH:'
202              TO WHERE-ERROR
203              PERFORM ERROR-PROCESS.
204          IF SQLSTATE NOT EQUAL TO NOT-FND-FIELD
205              PERFORM PRINT-DESCRIPTOR
206                  VARYING DESC-INDEX FROM 1 BY 1
207                  UNTIL DESC-INDEX>DESC-COUNT
208          DISPLAY ' '.
209 *
```

Lines 210 through 225

The `PRINT-DESCRIPTOR` subroutine uses the embedded dynamic SQL statement `GET DESCRIPTOR` to obtain values from the system descriptor area that the previously allocated descriptor **desc** identifies. It gets output values for the `:DESC-INDEX`, that serves as a counter, and for the `:RET-BUFFER`, the buffer that holds the data (`FNAME` and `LNAME`) that the `FETCH` statement returns.

The `PRINT-DESCRIPTOR` subroutine, a loop, displays the first and last names of the customers on the screen. The `DEMOCURSOR` selects a row from the **customer** table and puts the data from that row into the host variable `RET-BUFFER` (for `FNAME` and `LNAME`). `WITH NO ADVANCING` means that the program displays each row containing an `FNAME` and `LNAME` on the same line on the screen.

An IF statement returns the message ERROR ON GET DESCRIPTOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the PRINT-DESCRIPTOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```

210 *Subroutine to print a descriptor. Display data (names).
211 *
212 PRINT-DESCRIPTOR.
213     DISPLAY ' '.
214     IF COUNT-DESC IS EQUAL TO 1
215         DISPLAY 'EXECUTING GET DESCRIPTOR STATEMENTS'.
216     ADD 1 TO COUNT-DESC.
217     DISPLAY ' '.
218     EXEC SQL GET DESCRIPTOR 'desc' VALUE :DESC-INDEX
219         :RET-BUFFER=DATA END-EXEC.
220     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
221         MOVE 'ERROR ON GET DESCRIPTOR:'
222             TO WHERE-ERROR
223         PERFORM ERROR-PROCESS.
224     DISPLAY RET-BUFFER, ' ' WITH NO ADVANCING.
225 *
```

Lines 226 through 237

The CLOSE-CURSOR subroutine closes the cursor DEMOCURSOR using the embedded dynamic SQL statement CLOSE. It disassociates the cursor from the SELECT statement and stops the query process.

An IF statement returns the message ERROR ON OPEN CURSOR to the WHERE-ERROR variable when SQLSTATE does not equal "00000". When an error occurs during the execution of the CLOSE-CURSOR subroutine, the database server performs the ERROR-PROCESS subroutine.

```

226 *Subroutine to close a cursor.
227 *
228 CLOSE-CURSOR.
229     DISPLAY ' '.
230     DISPLAY 'EXECUTING CLOSE-CURSOR STATEMENT'.
231     DISPLAY ' '.
232     EXEC SQL CLOSE DEMOCURSOR END-EXEC.
233     IF SQLSTATE NOT EQUAL TO ZERO-FIELD
234         MOVE 'ERROR ON OPEN CURSOR:'
235             TO WHERE-ERROR
236         PERFORM ERROR-PROCESS.
237 *
```

Lines 238 through 248

The ERROR-PROCESS subroutine contains the process that counts SQLSTATE exceptions. That subroutine executes whenever an error occurs in one of the other subroutines. The DEMO3.ECO program stops running whenever the ERROR-PROCESS subroutine encounters an SQLCODE value not equal to ZERO.

SQLSTATE indicates the result of executing an SQL statement ("00000", "02000", or a value greater than "02000"). The GET DIAGNOSTICS NUMBER field contains the count of exceptions associated with the SQLSTATE code. The PERFORM UNTIL statement executes the EX-LOOP subroutine that displays an error message for each exception.

```
238 *Subroutine to check for exceptions.
239 *
240 ERROR-PROCESS.
241     DISPLAY WHERE-ERROR.
242     DISPLAY 'THE SQLSTATE CODE IS: ', SQLSTATE.
243     DISPLAY '*****'.
244     EXEC SQL GET DIAGNOSTICS :EX-COUNT=NUMBER END-EXEC.
245     PERFORM EX-LOOP UNTIL COUNTER IS GREATER THAN EX-COUNT.
246     IF SQLCODE NOT EQUAL TO ZERO
247         STOP RUN.
248 *
```

Lines 249 through 260

The EX-LOOP subroutine displays the exception number and the error message for each SQLSTATE exception.

```
249 *Subroutine to print exception messages.
250 *
251 EX-LOOP.
252     EXEC SQL
253         GET DIAGNOSTICS EXCEPTION :COUNTER
254         :MESS-TEXT=MESSAGE_TEXT
255     END-EXEC.
256     DISPLAY 'EXCEPTION ', COUNTER.
257     DISPLAY 'MESSAGE TEXT IS: ', MESS-TEXT.
258     DISPLAY '*****'.
259     ADD 1 TO COUNTER.
260 *
```


List of INFORMIX-ESQL/COBOL Routines

The table on the following page alphabetically lists the routines included with the INFORMIX-ESQL/COBOL library. It includes cross-references to the chapter that discusses the routine.



Warning: Informix encourages you to try the available examples for these routines using the MF COBOL/2 compiler. If you use the RM/COBOL-85 compiler, the example programs for the routines listed on the next page generate warnings and, in some cases, do not work at all. The examples for the following DATE routines, discussed in Chapter 3, do not compile correctly with RM/COBOL-85: ECO-DAT, ECO-DAY, ECO-DEF, ECO-FMT, ECO-JUL, ECO-LYR, ECO-MDY, ECO-STR, ECO-TDY.

List of Routines

Routine	Description	Page
ECO-DAI	Adds an INTERVAL string to a DATETIME string	3-38
ECO-DAT	Converts an internal format to a string	3-7
ECO-DAY	Returns the day of the week	3-10
ECO-DEF	Converts a string to an internal format	3-13
ECO-DSH	Converts a character string to lowercase	2-14
ECO-DSI	Subtracts an INTERVAL value from a DATETIME value	3-42
ECO-DTC	Determines the current DATETIME value	3-45
ECO-DTCVASC	Converts a specified format string to ANSI DATETIME format	3-47
ECO-DTS	Subtracts two DATETIME strings	3-52
ECO-DTTOASC	Converts a ANSI DATETIME string to a specified format	3-57
ECO-DTX	Extends a DATETIME value to a different qualifier	3-62
ECO-FFL	Returns a character string for a floating-point value	2-41
ECO-FIN	Returns a character string for an INTEGER value	2-43
ECO-FMT	Converts an internal format to a string	3-18
ECO-GST	Checks the SQLCODE OF SQLCA variable for three routines	2-20
ECO-IDI	Divides an INTERVAL value by an INTERVAL value	3-65
ECO-IDN	Divides an INTERVAL value by a numeric value	3-69
ECO-IMN	Multiplies an INTERVAL value by a numeric value	3-73
ECO-INCVASC	Converts a specified format string to ANSI INTERVAL format	3-77

(1 of 2)

Routine	Description	Page
ECO-INTOASC	Converts ANSI INTERVAL string to a specified ASCII format	3-82
ECO-INX	Extends an INTERVAL value to a different qualifier	3-87
ECO-IQU	Determines an INTEGER qualifier for a character-string qualifier	3-90
ECO-JUL	Returns the month, day, and year from an internal format	3-23
ECO-LYR	Determines whether it is a leap year	3-26
ECO-MDY	Returns an internal format from the month, day, and year	3-28
ECO-MSG	Converts an error message number into a message string	4-41
ECO-SIG	Allows the Informix library to perform signal handling	5-28
ECO-SQB	Sends the database server a request to stop processing	5-31
ECO-SQBCB	Registers a callback function	5-32
ECO-SQC	Checks the SQLCA record codes for three routines	2-21
ECO-SQD	Checks if database server is processing an SQL task	5-35
ECO-SQE	Terminates a database server process	5-37
ECO-SQS	Starts a database server process	5-41
ECO-SQU	Determines a character-string qualifier for an INTEGER qualifier	3-93
ECO-STR	Converts a string to an internal format	3-31
ECO-TDY	Returns the system date in an internal format	3-33
ECO-USH	Converts a character string to uppercase	2-17

(2 of 2)

Index

A

- Adding INTERVAL to DATETIME routine 3-38
- ALLOCATE DESCRIPTOR
 - statement, use in dynamic SQL 6-5
- Allocating memory
 - dynamically 6-10
 - using a system descriptor area 6-5 with ALLOCATE DESCRIPTOR 6-22
- ansi flag
 - and -xopen flag warnings 1-40
 - checking for Informix extensions 1-40
 - compiling programs with 1-36
- ANSI SQL Standards
 - for DATETIME values 3-36
 - for INTERVAL values 3-36
- ANSI-standard syntax, how to check for 1-39
- Array
 - and non-null SQL value 1-24
 - use within ESQL/COBOL statements 1-24

B

- Blobs
 - example program 2-29
 - programming with 2-25
 - use in dynamic SQL 2-27
- Break routine 5-31
- BYTE data type 2-25

C

- Callback procedure
 - CALLPROC example
 - program 5-22
 - cancelling SQL processing 5-13
 - CAN-QRY example program 5-18
 - checking SQL processing
 - status 5-13
 - comparison with other
 - procedures 5-16
 - described 5-13
 - discussion 5-13
 - example of 5-22
 - example of calling program 5-18
 - output from 5-25
 - process 5-14
 - rules for creating 5-17
 - status variable 5-17
 - using 5-13
- CHAR data type 2-11
- Character conversion routine
 - ECO-DSH 2-14
 - ECO-USH 2-17
- Character string
 - converting to lowercase 2-14
 - converting to uppercase 2-17
 - for INTEGER qualifier
 - routine 3-93
 - in date routine 3-18
- Client process 5-5
- Client-server architecture 5-4
- Client-server connections
 - connecting to a database
 - server 5-6
 - local 5-5
 - remote 5-5

- remote, using Relay Module 5-6
- sqlhosts file 5-7
- COBOL data types
 - MF COBOL/2 compiler PICTURE clauses 3-5
 - RM/COBOL 85 compiler PICTURE clauses 3-5
 - shown for CHAR routine arguments 2-12
 - shown for COBOL descriptions 2-12, 2-32, 3-5, 4-41
 - shown for DATE routine arguments 3-5
 - shown for ECO-MSG routine arguments 4-41
 - shown for routine arguments 2-32
- COBOL program structure
 - correspondence with SQL 2-4
 - embedding SQL statements 1-9
 - host variables 1-21
 - indicator variables 1-24
 - statement format 1-13
 - using SQL syntax 1-9
- COBOL routines
 - compiling 1-32
 - embedding SQL statements in 1-9
 - listed A-2
- COBOL statement format in source program 1-13
- Colon (:)
 - after main variable 1-26
 - before host variable 1-21
 - before indicator variable 1-26
- Comments, including in programs 1-17
- COMP equivalents
 - for CHAR routines 2-12
 - for ECO-MSG routine 4-42
 - for numeric-formatting routines 2-32
- Compiled program, running 1-42
- Compiler
 - creating an object file 1-32
 - diagnosing errors 1-22
 - linking library routines 2-13, 3-3, 5-27
 - syntax for esqlcobol command 1-33

- syntax for preprocessor naming options 1-36
- Compiler flag
 - ansi 1-36, 1-40, 1-41
 - bigB 1-36
 - comp89 1-36
 - e 1-33, 1-35
 - ED 1-36, 1-42
 - esqlout 1-36
 - EU 1-37, 1-42
 - I 1-37
 - icheck 1-28, 1-37, 1-40
 - local 1-37, 1-41
 - log 1-37, 1-41
 - n 1-33, 1-35
 - native 1-33
 - o 1-34
 - t 1-37
- testing for 1-29
- V 1-37
- w 1-37
- xopen 1-37, 1-40, 1-41
- Compiling
 - checking for ANSI-standard syntax 1-39
 - checking for missing indicator variables 1-40
 - checking the version number 1-37
 - displaying the processing steps 1-35
 - in X/Open mode 1-41
 - limiting the scope of cursor names 1-41
 - limiting the scope of statement ids 1-41
 - preprocessing only 1-35
 - redirecting errors and warnings 1-41
 - syntax for embedded SQL programs 1-32
 - with the esqlcobol command 1-32
 - with the -icheck flag 1-28
- Constants, statement type 6-16
- Conversion
 - how discrepancies are handled 2-10
 - INTERVAL to ASCII string, routine 3-82
 - of CHARACTER data 2-8

- of DATE data 2-9
- of DECIMAL data 2-9
- of FLOAT data 2-9
- of INTEGER data 2-8
- of SMALLFLOAT data 2-9
- of SMALLINT data 2-8
- to DATETIME string, routine
 - for 3-47
 - to INTERVAL string, routine for 3-77
- Converting dates, routines for 3-4
- COPY statements 1-32
- Current DATETIME routine 3-45
- Cursor names
 - in dynamic SQL 6-5
 - limiting the scope with the -local option 1-41

D

- Data conversion
 - among data types 2-10
 - DATE 2-9
 - DECIMAL 2-9
 - FLOAT 2-9
 - problems 2-10
 - SMALLFLOAT 2-9
- DATA field 6-12
- Data types
 - BYTE 2-25
 - CHAR 2-11
 - correspondence 2-4
 - DATE 3-3
 - DATETIME 3-35
 - declaration for GET DESCRIPTOR 6-7
 - declaration for SET DESCRIPTOR 6-7
 - INTERVAL 3-35
 - TEXT 2-25
 - type conversion 2-10
 - VARCHAR 2-22
- Database server connection
 - using DISCONNECT statement 5-12
- Database server connections
 - current 5-9
 - default 5-10

- dormant 5-9
- establishing 5-11
- explicit 5-8
- implicit 5-9
- multiple 5-8
- specific 5-11
- terminating 5-12
- types 5-8
- using CONNECT TO DEFAULT statement 5-11
- using CONNECT TO statement 5-11
- using CREATE DATABASE statement 5-12
- using DATABASE statement 5-12
- using DISCONNECT ALL statement 5-12
- using DISCONNECT CURRENT statement 5-12
- using DISCONNECT DEFAULT statement 5-12
- using DROP DATABASE statement 5-12
- using ECO-SQE routine 5-12
- using ECO-SQS routine 5-12
- using SET CONNECTION statement 5-12
- using START DATABASE statement 5-12
- Database server control routine
 - break 5-31, 5-32, 5-35
 - check SQL processing status 5-35
 - ECO-SIG 5-28
 - ECO-SQB 5-31, 5-37
 - ECO-SQBCB 5-32
 - ECO-SQD 5-35
 - ECO-SQS 5-41
 - exit 5-37
 - register callback procedure 5-32
 - signal handling 5-28
 - start 5-41
 - start a connection 5-41
 - stop SQL processing 5-31
 - terminate all connections 5-37
- Date conversion routine 3-7
- DATE data type 3-3
- DATE manipulation routine
 - ECO-DAT 3-7
 - ECO-DAY 3-10

- ECO-DEF 3-13
- ECO-FMT 3-18
- ECO-JUL 3-23
- ECO-LYR 3-26
- ECO-MDY 3-28
- ECO-STR 3-31
- ECO-TDY 3-33
- Date string conversion routine 3-13
- DATETIME data type 3-35
- DATETIME manipulation routine
 - ECO-DAI 3-38
 - ECO-DSI 3-42
 - ECO-DTC 3-45
 - ECO-DTCVASC 3-47
 - ECO-DTS 3-52
 - ECO-DTTOASC 3-57
 - ECO-DTX 3-62
 - ECO-IQU 3-90
 - ECO-SQU 3-93
- DATETIME value, extending to a different qualifier 3-62
- Day of week routine 3-10
- DBANSIWARN environment variable 1-39
- DBTIME environment variable
 - when to use 3-36
- DEALLOCATE DESCRIPTOR statement, use in dynamic SQL 6-5
- Declaration
 - and FILLER keyword 1-16
 - of dynamic cursor names 1-41
 - of dynamic statement ids 1-41
 - of group items and arrays 1-23
 - of host variables 1-21
 - of indicator variables 1-26
 - of non-host program variables 1-22
 - of variables outside the DECLARE section 1-23
- DEFINE preprocessor instruction 1-29
- Defining values while preprocessing 1-42
- DEMO1.ECO program
 - complete example shown 1-46
 - line-by-line explanation 1-48
- DEMO2.ECO program
 - complete example shown 6-35

- line-by-line explanation 6-39
- DEMO3.ECO program
 - complete example shown 6-49
 - line-by-line explanation 6-54
- Demonstration database
 - DEMO1.ECO program 1-46
 - DEMO2.ECO program 6-35
 - DEMO3.ECO program 6-49
- example programs 6-34
- DESCRIBE statement, use in dynamic SQL 6-5
- Descriptor
 - and system descriptor area 6-5
 - in demo programs 6-34
- Dollar sign (\$)
 - displaying a literal 2-34
 - floating 2-34
- Downshifting character strings
 - with ECO-DSH 2-14
- Dynamic management
 - implemented in two demo programs 6-34
 - SQL statements and techniques 6-7
- Dynamic SQL
 - allocating memory 6-10
 - and backward compatibility in programs 1-41
 - and blobs 2-27
 - and stored procedures 6-29
 - creating stored procedures 6-30
 - descriptor statements 6-5
 - executing stored procedures 6-29
 - four statement types described 6-8
 - localizing dynamic cursor names 1-37
 - localizing dynamic statement identifier names 1-37
 - non-parameterized non-SELECT statements 6-27
 - non-parameterized SELECT statements 6-8, 6-24
 - parameterized non-SELECT statements 6-26
 - parameterized SELECT statements 6-8, 6-19
 - program examples 6-34
 - programming with 6-4

- sample program using stored procedures 6-30
- SELECT statements that receive WHERE-clause values at run time 6-19
- SELECT statements where select - list values are determined at run time 6-24
- statement type constants for 6-16
- statements that do not receive values at run time 6-27
- statements that receive values at run time 6-26
- use in programming 6-4
- using a system descriptor area 6-10
- when to use 6-8
- working with a system descriptor area 6-6

E

- ECO-DAI routine 3-38
- ECO-DAT routine 3-7
- ECO-DAY routine 3-10
- ECO-DEF routine 3-13
- ECO-DSH routine 2-14
- ECO-DSI routine 3-42
- ECO-DTC routine 3-45
- ECO-DTCVASC routine 3-47
- ECO-DTS routine 3-52
- ECO-DTTOASC routine 3-57
- ECO-DTX routine 3-62
- ECO-FFL routine 2-41
- ECO-FIN routine 2-43
- ECO-FMT routine 3-18
- ECO-GST routine 2-20
- ECO-IDI routine 3-65
- ECO-IDN routine 3-69
- ECO-IMN routine 3-73
- ECO-INCVASC routine 3-77
- ECO-INTOASC routine 3-82
- ECO-INX routine 3-87
- ECO-IQU routine 3-90
- ECO-JUL routine 3-23
- ECO-LYR routine 3-26
- ECO-MDY routine 3-28
- ECO-MSG routine 4-41

- ECO-SIG routine 5-28
- ECO-SQB routine 5-31, 5-32, 5-35
- ECO-SQBCB routine 5-32
- ECO-SQC routine 2-21
- ECO-SQD routine 5-35
- ECO-SQE routine 5-37
- ECO-SQS routine 5-41
- ECO-SQU routine 3-93
- ECO-STR routine 3-31
- ECO-TDY routine 3-33
- ECO-USH routine 2-17
- ELIF preprocessor instruction 1-30
- ELSE preprocessor instruction 1-30
- Embedded SQL statements
 - compiling and preprocessing 1-32
 - in COBOL routines 1-9
- END-EXEC
 - in BEGIN DECLARE section 1-21
 - with SQL statement 1-17
- ENDIF preprocessor instruction 1-30
- Environment variables
 - DBANSIWARN 1-39
 - DBTIME 1-5
 - INFORMIXCOBTYPE 1-5
 - to specify compiler 1-5
 - to specify location of run-time library 1-5
 - using INFORMIXCOBSTORE 2-7
 - what to set 1-5
- Error handling
 - and the WHENEVER statement 1-20, 4-29
 - checking for errors using GET DIAGNOSTICS 4-25
 - checking for errors using in-line code 4-26
 - example program using GET DIAGNOSTICS 4-45
 - obtaining diagnostic information after an SQL statement 4-4
 - overview 4-3
 - result codes
 - error 4-24
 - no data found, end of data 4-22
 - success 4-22
 - success with warning 4-22

- SQL statement result codes 4-21
- using GET DIAGNOSTICS 4-4
- Error message conversion routine, ECO-MSG 4-41
- Errors
 - after a GET DIAGNOSTICS statement 4-25
 - after a PREPARE statement 4-24
 - after an EXECUTE statement 4-24
 - automatically checking for 4-29
 - redirecting with the -log option 1-41
- esqlcobol
 - command-line syntax 1-33
 - compiling with the shell script 1-32
 - linking run-time routines 2-13, 3-3, 5-27
 - preprocessor naming options 1-36
 - using to preprocess, compile, and link 1-34
- ESQL/COBOL library
 - list of all routines A-2
 - list of CHAR routines 2-13
 - list of DATE routines 3-4
 - list of DATETIME and INTERVAL routines 3-35
 - list of numeric-formatting routines 2-31
- Example programs
 - DEMO1.ECO 1-46
 - DEMO2.ECO 6-35
 - DEMO3.ECO 6-49
 - in demonstration database 6-34
- EXEC SQL
 - in BEGIN DECLARE section 1-21
 - with SQL statement 1-17
- EXECUTE IMMEDIATE statement, use in dynamic SQL 6-28
- EXECUTE INTO
 - replacing PREPARE, OPEN, and FETCH 6-33
 - using in dynamic program 6-33
 - using with stored procedures 6-33
- EXECUTE statement
 - errors after 4-24
 - in dynamic SQL 6-27
- Exit routine 5-37

Expressions
 formatting 2-31
 routines for formatting numeric
 ones 2-31
 table of numeric format
 strings 2-33

F

File extension
 .cbl 1-34
 .cob 1-33, 1-34
 .eco 1-32, 1-34
 .exe 1-34
 .INT 1-34
FILLER keyword 1-16
Formatting numeric expressions
 examples 2-34
 overview 2-31
 strings 2-33
 valid characters 2-33
Formatting routines, numeric 2-31
Function calls
 CHAR type routines listed 2-13
 DATE type routines listed 3-3
 DATETIME type routines
 listed 3-35
 INTERVAL type routines
 listed 3-35
 Numeric-formatting routines
 listed 2-31

G

GET DESCRIPTOR statement, use
 in dynamic SQL 6-6
GET DIAGNOSTICS
 CLASS_ORIGIN field 4-5
 CONNECTION_NAME field 4-6
 detecting and handling errors 4-4
 diagnosing multiple errors 4-16
 exception information 4-5
 MESSAGE_LENGTH field 4-6
 MESSAGE_TEXT field 4-6
 MORE field 4-5
 multiple error conditions 4-16

NUMBER field 4-5
 overview 4-4
 RETURNED_SQLSTATE field 4-5
 ROW_COUNT field 4-5
 SERVER_NAME field 4-6
 statement information 4-4
 SUBCLASS_ORIGIN field 4-5
 usage 4-7
 using full error checking 4-45
Group items declared as host
 variables 1-23

H

Host variables
 and arrays 1-24
 and group items 1-23
 and VALUE clauses 1-22
 choosing data types for 2-4
 COBOL features not recognized
 in declarations 1-22
 declared with COBOL initializer
 expressions 1-22
 definition of 1-21
 how to declare 1-21
 in parameterized SELECT
 statement 6-20
 preceded by colon 1-21
 use of VARCHAR in
 programming 2-23
 used in place of a constant 1-12
 what is allowed in PICTURE
 clause 2-5
 what is not allowed in PICTURE
 clause 2-5
 with parameterized
 statements 6-26

I

icheck flag, compiling programs
 with 1-37
IDATA field 6-12
IFDEF preprocessor
 instruction 1-29
IFNDEF preprocessor
 instruction 1-29

ILENGTH field 6-12
INCLUDE preprocessor instruction
 compared to COBOL COPY
 statement 1-32
 definition of 1-29
 syntax 1-31
INDICATOR field 6-12
INDICATOR keyword, and
 indicator variable 1-26
Indicator variables
 and associated host
 variables 1-24, 1-28
 and INDICATOR keyword 1-26
 and null values 1-27
 checking for missing ones with -
 icheck compiler option 1-40
 defined 1-24
 how to declare 1-26
 main variable 1-25
 truncation of 1-24, 1-25
 used in place of a constant 1-12
 with null and not null values 1-24
INFORMIXCOBSTORE
 environment variable 2-7
INTEGER date conversion
 routine 3-31
INTEGER qualifier for character
 string routine 3-90
Interactive programs and dynamic
 SQL 6-4
INTERVAL added to DATETIME
 routine 3-38
INTERVAL data type 3-35
INTERVAL manipulation routine
 ECO-DAI 3-38
 ECO-DSI 3-42
 ECO-IDI 3-65
 ECO-IDN 3-69
 ECO-IMN 3-73
 ECO-INCVASC 3-77
 ECO-INTOASC 3-82
 ECO-INX 3-87
 ECO-IQU 3-90
 ECO-SQU 3-93

INTERVAL value
divided by INTERVAL value 3-65
divided by numeric value 3-69
extended to a different
qualifier 3-87
multiplied by a numeric
value 3-73
ITYPE field 6-12

J

Julian date routine 3-23

L

Leap year routine 3-26
LENGTH field 6-12
Library routines

complete list A-2
ECO-DAI 3-38
ECO-DAT 3-7
ECO-DAY 3-10
ECO-DEF 3-13
ECO-DSH 2-14
ECO-DSI 3-42
ECO-DTC 3-45
ECO-DTCVASC 3-47
ECO-DTS 3-52
ECO-DTTOASC 3-57
ECO-DTX 3-62
ECO-FFL 2-41
ECO-FIN 2-43
ECO-FMT 3-18
ECO-GST 2-20
ECO-IDI 3-65
ECO-IDN 3-69
ECO-IMN 3-73
ECO-INCVASC 3-77
ECO-INTOASC 3-82
ECO-INX 3-87
ECO-IQU 3-90
ECO-JUL 3-23
ECO-LYR 3-26
ECO-MDY 3-28
ECO-MSG 4-41
ECO-SIG 5-28
ECO-SQB 5-31, 5-32, 5-35
ECO-SQC 2-21

ECO-SQE 5-37
ECO-SQS 5-41
ECO-SQU 3-93
ECO-STR 3-31
ECO-TDY 3-33
ECO-USH 2-17
included in ESQ/COBOL
library 2-13, 2-31, 3-4, 3-35,
5-27, A-2
using esqlcobol 2-13, 3-3
Linking, with preprocessing and
compiling 1-34
local flag, compiling programs
with 1-37, 1-41
Localized DATETIME format
routine 3-57
log flag, compiling programs
with 1-37, 1-41

M

Manipulating DATE types
ECO-DAT routine 3-7
ECO-DAY routine 3-10
ECO-DEF routine 3-13
ECO-FMT routine 3-18
ECO-JUL routine 3-23
ECO-LYR routine 3-26
ECO-MDY routine 3-28
ECO-STR routine 3-31
ECO-TDY routine 3-33
Manipulating DATETIME types
ECO-DAI routine 3-38
ECO-DSI routine 3-42
ECO-DTC routine 3-45
ECO-DTCVASC routine 3-47
ECO-DTS routine 3-52
ECO-DTTOASC routine 3-57
ECO-DTX routine 3-62
ECO-IQU routine 3-90
ECO-SQU routine 3-93
Manipulating INTERVAL types
ECO-DAI routine 3-38
ECO-DSI routine 3-42
ECO-IDI routine 3-65
ECO-IDN routine 3-69
ECO-IMN routine 3-73
ECO-INCVASC routine 3-77

ECO-INTOASC routine 3-82
ECO-INX routine 3-87
Memory allocation
and the system descriptor
area 6-10
in dynamic SQL statements 6-5
Micro Focus COBOL
customizing a run-time
program 1-9
numbers for subroutine
names 1-17
run-time program for 1-9
standard integer in 3-5
storage allocation 2-6
using BINARY or COMP data 2-6
Month, day, year routine 3-28

N

NAME field 6-13
native flag, compiling programs
with 1-33
Non-parameterized non-SELECT
statements 6-9, 6-27
Non-parameterized SELECT
statements 6-8, 6-24
Null values
and indicator variables 1-27
and the -icheck flag 1-28
NULLABLE field 6-13
Numeric expressions
example formats 2-34
formatting 2-31
valid characters 2-33
Numeric-formatting routine
ECO-FFL 2-41
ECO-FIN 2-43

O

Option
-ansi 1-36, 1-40, 1-41
-bigB 1-36
-comp89 1-36
-e 1-33, 1-35
-ED 1-36, 1-42
-esqlout 1-36
-EU 1-37, 1-42

- I 1-37
- icheck 1-28, 1-37, 1-40
- local 1-37, 1-41
- log 1-37, 1-41
- n 1-33, 1-35
- native 1-33
- o 1-34
- t 1-37
- V 1-37
- w 1-37
- xopen 1-37, 1-40, 1-41

P

Parameterized non-SELECT statements

- description of 6-9, 6-26
- using a system descriptor area 6-27
- using host variables 6-26

Parameterized SELECT statements

- description of 6-8, 6-19
- using a system descriptor area 6-21
- using host variables 6-20

PICTURE clause

- for DATE data type columns 3-4
- for ECO-MSG arguments 4-42
- relationship to SQL data types 2-7
- what is allowed for host variables 2-5
- what is not allowed for host variables 2-5
- where to declare 2-5

PRECISION field 6-13

PREPARE statement

- errors after 4-24
- in dynamic SQL 6-27
- missing WHERE signalled 4-20

Preprocessing

- checking for ANSI-standard syntax 1-39
- checking for missing indicator variables 1-40
- including alternative SQLCA header files 1-38
- with compiling and linking 1-34
- without compiling 1-35

Preprocessor

- conditional compilation of ESQL/COBOL statements 1-29
- defining and undefining values 1-42
- detecting tabs 1-16
- embedding SQL/COBOL routines 1-32
- redirecting errors and warnings 1-41
- search sequence for included files 1-31

- statements handled 1-29
- subroutine names 1-17
- supported instructions 1-29
- syntax for options 1-36

Preprocessor instructions

- DEFINE 1-29

- ELIF 1-30
- ELSE 1-30
- ENDIF 1-30
- IFDEF 1-29
- IFDEF 1-29
- INCLUDE 1-29
- UNDEF 1-29

Procedure, see Callback Procedure 5-13

Processing

- and INCLUDE statement 1-32
- displaying steps without executing 1-35
- sequence of occurrence 1-34

Program

- compiling 1-32
- example, DEMO1.ECO 1-46
- example, DEMO2.ECO 6-35
- example, DEMO3.ECO 6-49
- using the SQLCODE variable 4-21

Programming

- compiling your programs 1-32
- declaring group items and arrays 1-23
- declaring host variables 1-21
- declaring indicator variables 1-26
- embedding SQL statements 1-9
- error handling 1-19
- full error checking using GET DIAGNOSTICS 4-45

- including comments 1-17
- indicator variables and null values 1-27
- reserved words and conventions 1-17
- the COBOL statement format 1-13
- using host variables 1-21
- using indicator variables 1-24
- using the preprocessor 1-29
- using the SQLCA record 1-19
- with dynamic SQL statements 6-4

Q

Question mark (?)

- in dynamic SQL statements 6-4
- in parameterized SELECT statements 6-19
- in PICTURE clause 2-12, 2-32, 4-42
- use in programming 6-4

R

Reserved words

- finding with the -ansi flag 1-40
- in COBOL 1-17
- in SQL statements 1-17

Routines

- complete list A-2
- database server 5-27
- ECO-DAI 3-38
- ECO-DAT 3-7
- ECO-DAY 3-10
- ECO-DEF 3-13
- ECO-DSH 2-14
- ECO-DSI 3-42
- ECO-DTC 3-45
- ECO-DTCVASC 3-47
- ECO-DTS 3-52
- ECO-DTTOASC 3-57
- ECO-DTX 3-62
- ECO-FIN 2-43
- ECO-FMT 3-18
- ECO-GST 2-20
- ECO-IDI 3-65
- ECO-IDN 3-69
- ECO-IMN 3-73
- ECO-INCVASC 3-77

- ECO-INTOASC 3-82
- ECO-INX 3-87
- ECO-IQU 3-90
- ECO-JUL 3-23
- ECO-LYR 3-26
- ECO-MDY 3-28
- ECO-SIG 5-28
- ECO-SQB 5-31, 5-32, 5-35
- ECO-SQC 2-21
- ECO-SQE 5-37
- ECO-SQS 5-41
- ECO-SQU 3-93
- ECO-STR 3-31
- ECO-TDY 3-33
- ECO-USH 2-17
 - that work with the database server 5-27
- Running a compiled program 1-42
- Run-time program
 - for Micro Focus COBOL 1-9
 - for Ryan-McFarland COBOL 1-7
- Ryan-McFarland COBOL compiler flag for 1-37
- customizing a run-time program 1-7
- run-time program for 1-7
- standard integer in 3-5

S

- Sample program
 - DEMO1.ECO 1-45
 - DEMO2.ECO 6-35
 - DEMO3.ECO 6-49
- SCALE field 6-13
- SELECT statement
 - non-parameterized 6-8, 6-24
 - parameterized 6-8, 6-19
 - that receives WHERE-clause values at run time 6-19
 - where select-list values are determined at run time 6-24
- Select-list values, determined at run time 6-24
- Server control routine
 - break 5-31
 - exit 5-37
 - start 5-41
- Server process 5-5
- SET DESCRIPTOR statement, use in dynamic SQL 6-6
- Signal-handling routine, ECO-SIG 5-28
- SQL
 - allocating space in memory 6-5
 - and END-EXEC keyword 1-17
 - and EXEC SQL keywords 1-17
 - choosing data types for host variables 2-4
 - conversion of data types 2-7
 - correspondence with COBOL data types 2-4
 - embedding statements in COBOL routines 1-9
 - four types of dynamic statements 6-8
 - use in COBOL 1-9
- SQL processing
 - breaking 5-31
 - checking status of 5-35
- SQL statements
 - dynamic 6-3
 - INTEGER type constants for 6-16
- SQLCA record
 - checking for routines without STATUS 2-20, 2-21
 - signaling truncation 1-25
- SQLCODE variable, using in programs 4-21
- sqllda structure. *See* system descriptor area. 6-10
- sqlhosts file, setting up 5-7
- SQLSTATE variable
 - class code 4-9
 - error status code 4-9
 - structure 4-9
 - subclass code 4-9
 - using in applications 4-14
 - using with GET DIAGNOSTICS 4-9
 - valid codes, list of 4-11
- Start routine 5-41
- Statement ids
 - in dynamic SQL 6-5
 - limiting the scope with the -local option 1-41

- Statement type constants in dynamic SQL 6-16
- Statements used in dynamic SQL 6-3
- Storage allocation for Micro Focus COBOL 2-6
- Stored procedures
 - creating 6-30
 - executing dynamically 6-30
 - sample program 6-30
 - using EXECUTE INTO statement within 6-33
 - using with dynamic SQL 6-29
- Strings, formatting numeric 2-33
- Subtracting two DATETIME values routine 3-52
- System date routine 3-33
- System descriptor area
 - allocating space in memory 6-5
 - and descriptor statements 6-5
 - data types for 6-6
 - fields defined 6-12
 - using 6-10
 - values for TYPE and ITYPE fields 6-13
 - with non-parameterized non-SELECT statements 6-27
 - with parameterized statements 6-27

T

- TEXT data type 2-25
- Truncation
 - avoiding with SQL and COBOL data types 2-7
 - conversion problems 2-10
 - of indicator variable 1-25
 - of SQL variable 1-24
 - signaled in SQLCA record 1-25
- TYPE field 6-13

U

- UNDEF preprocessor instruction 1-29
- Undefined values while preprocessing 1-42

UPDATE statement, missing
WHERE signalled 4-20
Upshifting character strings with
ECO-USH 2-17

V

VALUE clauses 1-22

Values

- defining and undefining with
preprocessor options 1-42
- determined at run time in
SELECT statement 6-24
- holding standard date 6-7
- not received at run time by non-
SELECT statement 6-27
- received at run time by non-
SELECT statement 6-26
- received at run time by SELECT
statement 6-19
- returned from a FETCH
statement 6-7
- returned from a query 6-24
- storing julian dates 6-7
- that change at run time 6-4

VARCHAR

- data type 2-22
- host variables, programming
with 2-23

Variable

- character data type 2-22
- host 1-21
- indicator 1-24
- SQLCODE 4-21
- truncation 1-24

Version number, how to check 1-37

W

WARN example program 4-36

Warnings

- and SQLWARN OF SQLCA 1-40,
2-11
- and the ECO-GST routine 2-20
- and the ECO-SQC routine 2-21
- checking for
using sqlca structure 4-38

- checking with the
DBANSIWARN environment
variable 1-39
- diagnosing with GET
DIAGNOSTICS
statement 4-35
- generated for both -ansi and -
xopen flags 1-40
- generated for various SQL
statements 4-35
- generating with the -ansi
flag 1-36, 1-40
- redirecting with the -log
option 1-41
- sample program 4-36
- sending to specified file 1-37
- trapping with the WHENEVER
statement 1-20
- using the SQLWARNING
keyword 4-38
- WARN example program 4-36
- WHENCHK example program 4-31
- WHENEVER STATEMENT
executing a call 4-29
- WHENEVER statement
continuing execution 4-29
- discussion 4-29
- example program 4-31
- executing a call 4-29
- stopping execution 4-29
- trapping errors 4-29
- trapping warnings 4-29
- use 4-29
- using GOTO within 4-29
- using PERFORM within 4-29
- WHENCHK example
program 4-31
- WHENEVER statement, used to
trap errors and warnings 4-29
- WHERE-clause values, received at
run time 6-19

X

xopen flag

- and -ansi flag warnings 1-40
- compiling programs with 1-37

X/Open mode, compiling 1-41

