HP NonStop SQL/MP Programming Manual for COBOL

Abstract

This manual documents the programming interface to HP NonStop™ SQL/MP for COBOL. It is intended for application programmers who are embedding SQL statements and directives in COBOL programs.

Product Version

NonStop SQL/MP G06 and H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.20 and all subsequent G-series RVUs, and D46.00 and all subsequent D-series RVUs until otherwise indicated by its replacement publications.

Part Number	Published
529758-003	August 2012

Document History

Part Number	Product Version	Published
429326-002	NonStop SQL/MP G06	May 2003
429326-003	NonStop SQL/MP G06	December 2003
429326-004	NonStop SQL/MP G06	December 2004
529758-001	NonStop SQL/MP G06	April 2005
529758-002	NonStop SQL/MP G06 and H01	August 2010
529758-003	NonStop SQL/MP G06 and H01	August 2012

Legal Notices

HP NonStop SQL/MP Programming Manual for COBOL

Index   Examples   Figures   Tables

E. Writing Pathway Servers (continued)

Figures

Tables

What’s New in This Manual

Manual Information

Abstract

This manual documents the programming interface to HP NonStop™ SQL/MP for COBOL. It is intended for application programmers who are embedding SQL statements and directives in COBOL programs.

Product Version

NonStop SQL/MP G06 and H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.20 and all subsequent G-series RVUs, and D46.00 and all subsequent D-series RVUs until otherwise indicated by its replacement publications.

Part Number	Published
529758-003	August 2012

Document History

Part Number	Product Version	Published
429326-002	NonStop SQL/MP G06	May 2003
429326-003	NonStop SQL/MP G06	December 2003
429326-004	NonStop SQL/MP G06	December 2004
529758-001	NonStop SQL/MP G06	April 2005
529758-002	NonStop SQL/MP G06 and H01	August 2010
529758-003	NonStop SQL/MP G06 and H01	August 2012

New and Changed Information

Changes to the 529758-003 manual:

- Added -Wsqlconnect compiler option in -Wsqlconnect on page 6-21.
- Added -HP_NSK_CONNECT_MODE environment variable option in HP_NSK_CONNECT_MODE on page 6-22.

Changes to the 529758-002 manual:

- Added applicability note for SQL integer data types on page 2-5 and page 2-8.
- Modified the ALTER TABLE and ALTER INDEX information under Causes of SQL Error 8204 (Lost Open Error) on page 4-2.

Changes to the 529758-001 Manual

- Changed the manual title from *HP NonStop SQL/MP Programming Manual for COBOL85* to *HP NonStop SQL/MP Programming Manual for COBOL*. The manual now uses the term *COBOL85* to refer only to the COBOL85 compiler. The COBOL85 language is now referred to as *COBOL*. The HP COBOL compiler means both the COBOL85 compiler and the NMCOBOL compiler.
- Updated information related to process access:
  - On page 2-14, for the INVOKE directive
  - On page 4-4, for the SELECT statement
  - On page 4-6, for the INSERT statement
  - On page 4-8, for the UPDATE statement
  - On page 4-10, for the DELETE statement
  - On page 4-13, for the OPEN CURSOR statement
  - On page 4-17, for the FETCH statement
  - On page 4-19, for a multirow SELECT statement
  - On page 4-20, for the UPDATE statement used with a cursor
  - On page 4-22, for the DELETE statement with a cursor
  - On page 6-35, for the UPDATE STATISTICS statement
- Added information about compiling HP COBOL programs in the PC environment and using TNS compilation tools in Section 6, Explicit Program Compilation.
- Added information about process access privileges for an SQL statement under Required Access Authority on page 7-1.
- Added information about executing an SQL program from a COBOL program on a TNS/R system Using the CLU_PROCESS_CREATE_ Routine on page 7-4.
- Changed the real memory from 2 KB to 16 KB pages under Guidelines for Memory Use on page B-6.

About This Manual

This manual describes the NonStop SQL/MP programmatic interface for HP COBOL for NonStop systems. Using this interface, a COBOL program can access a NonStop SQL/MP database by using embedded SQL statements and directives. The HP COBOL compiler means both the COBOL85 compiler and the NMCOBOL compiler.

Who Should Read This Guide

This manual is intended for application programmers who are embedding SQL/MP statements and directives in a COBOL program. The reader should be familiar with COBOL, NonStop SQL/MP terms and concepts (as described in the *Introduction to NonStop SQL/MP*), and the HP NonStop operating system.

Related Manuals

The related manuals that an application programmer might find useful are:

- NonStop SQL/MP library
- Program development manuals
- Guardian system manuals

Table i describes the manuals in the NonStop SQL/MP library.

Table i. NonStop SQL/MP Library (page 1 of 2)

Manual	Description
Introduction to NonStop SQL/MP	Introduces the NonStop SQL/MP relational database management system.
SQL/MP Reference Manual	Describes the NonStop SQL/MP language elements, including expressions, functions, commands, statements, SQLCI utilities and commands, and report writer commands. (This manual is the printed version of online help.)
SQL/MP Messages Manual	Describes error and warning numbers and messages returned by NonStop SQL, the SQL file system, and FastSort.
SQL/MP Query Guide	Describes how to retrieve and modify data in a NonStop SQL/MP database and how to analyze and improve query performance.
SQL/MP Version Management Guide	Describes the rules governing version management for the NonStop SQL/MP software, catalogs, objects, messages, programs, and data structures.

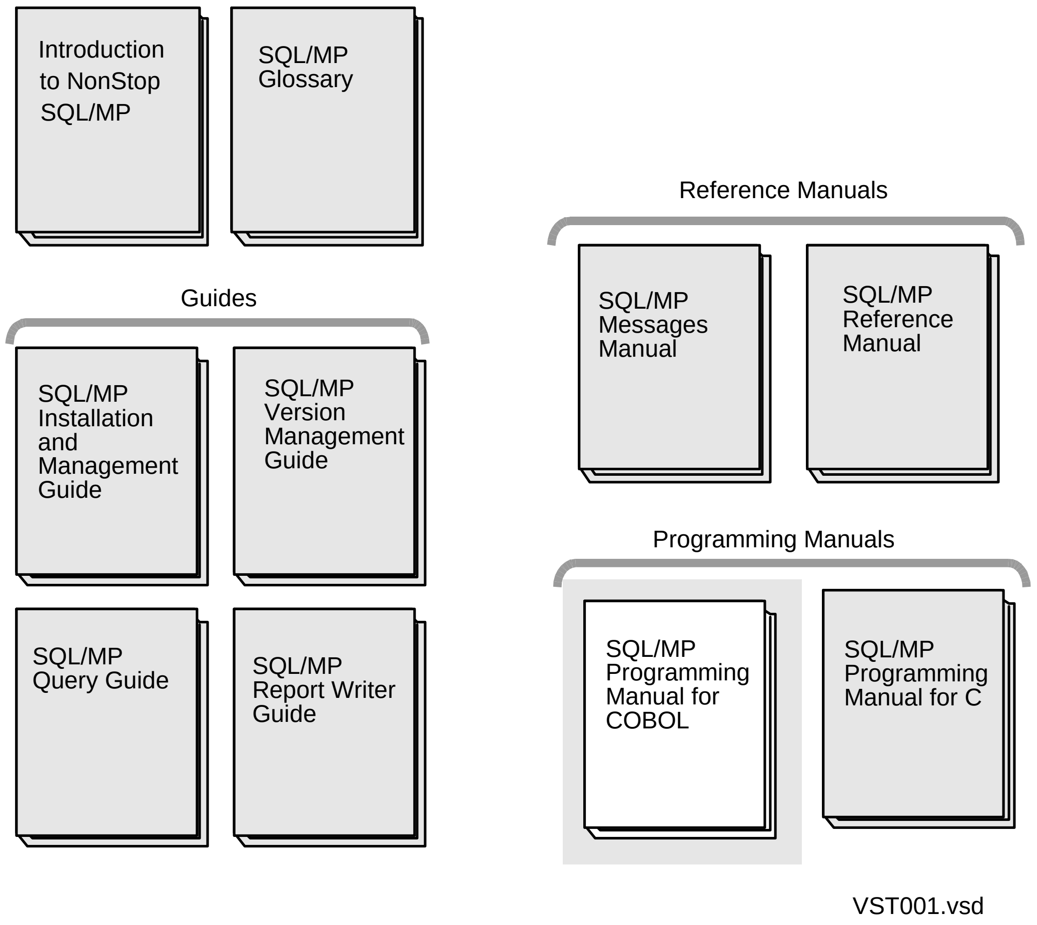
* C30.07 manual; does not include D-series information.

Table i. NonStop SQL/MP Library (page 2 of 2)

SQL/MP Installation and Management Guide	Describes how to plan, install, create, and manage a NonStop SQL/MP database and SQL programs.
SQL/MP Report Writer Guide	Describes how to use report writer commands and SQLCI options to design and produce reports.
SQL/MP Programming Manual for C *SQL/MP Programming Manual for COBOL* *SQL Programming Manual for Pascal* * *SQL Programming Manual for TAL* *	Describes the NonStop SQL/MP programmatic interface for C, COBOL, Pascal, and TAL applications.

* C30.07 manual; does not include D-series information.

Figure i. NonStop SQL/MP Library


Introduction to NonStop SQL/MP
SQL/MP Glossary
Reference Manuals
SQL/MP Messages Manual
SQL/MP Reference Manual
Guides
SQL/MP Installation and Management Guide
SQL/MP Version Management Guide
SQL/MP Query Guide
SQL/MP Report Writer Guide
Programming Manuals
SQL/MP Programming Manual for COBOL
SQL/MP Programming Manual for C
VST001.vsd

In addition to the NonStop SQL/MP library, program development, Guardian, and HP NonStop Open System Services (OSS) manuals described in these tables can be useful to a COBOL programmer.

Table ii. Program Development Manuals

Manual	Description
COBOL85 for NonStop Systems Manual	Describes the HP implementation of COBOL, including the statement syntax, run-time library, program execution environment, HP extensions, and how to use HP COBOL.
CRE Programmer's Guide	Describes the Common Run-Time Environment (CRE) and how to write and run mixed-language programs.
Inspect Manual	Describes the Inspect program, an interactive source-level or machine-level debugger that enables you to interrupt and resume program execution and to display and modify variables.
CROSSREF Manual	Describes the `CROSSREF` program, which produces a cross-reference listing of selected identifiers in an application.
Binder Manual	Describes the Binder program, an interactive linker that enables you to examine, modify, and combine compilation units (object files) and to generate load maps and cross-reference listings.
Accelerator Manual	Describes the Accelerator for HP TNS/R systems for optimizing the program-file object code.
Debug Manual	Describes the Debug program, an interactive machine-level debugger.

Table iii. Guardian Manuals

Manual	Description
Guardian Application Conversion Guide	Describes how to convert C, COBOL, Pascal, TAL, and TACL applications to use the extended features of the NonStop OS.
Guardian Procedure Calls Reference Manual	Describes the syntax for Guardian procedure calls.
Guardian Programmer's Guide	Describes how to use Guardian procedure calls from an application to access operating system services.
Guardian Procedure Errors and Messages Manual	Describes error codes, error lists, system messages, and trap numbers for Guardian system procedures.

Table iv. Open System Services (OSS) Manuals

Manual	Description
Open System Services Programmer's Guide	Describes how to use the OSS application programming interface to the operating system.
Open System Services Shell and Utilities Reference Manual	Describes the syntax and semantics for using the OSS shell and utilities.
Open System Services System Calls Reference Manual	Describes the syntax and programming considerations for using OSS system calls.

Notation Conventions

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]
```

```
K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
```

```
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                  !i
                         , error        ) ;            !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                    !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length         !i:i
                            , filename2:length ) ;     !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                      !i
                        , [ filename:maxlen ] ) ;      !o:i
```

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the HP COBOL environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Introduction

NonStop SQL/MP is the HP relational database management system (RDBMS) that uses SQL to define and manipulate data in an SQL/MP database. You can run SQL statements interactively by using the SQL/MP conversational interface (SQLCI) or programmatically by embedding SQL statements and directives in a host-language program written in COBOL, C, Pascal, or TAL.

This manual describes the programmatic interface to SQL/MP for COBOL programs.

This section discusses:

- Advantages of Using Embedded SQL Statements
- Development of a COBOL Program on page 1-2
- Dynamic SQL Operations on page 1-6
- SQL/MP Version Management on page 1-7
- COBOL in the Open System (OSS) Environment on page 1-7
- Effect on Conformance to ISO/ANSI Standards on page 1-8

Advantages of Using Embedded SQL Statements

Embedding SQL statements and directives in a COBOL program to access an SQL/MP database has these advantages:

- A high-level, efficient database language—You can code a request to access the database by using SQL statements. The SQL optimizer then generates an efficient plan to perform your request.
- Insulation against database changes—If a database administrator modifies an SQL/MP database (for example, adds a column to a table), the change does not affect the logic of your program.
- COBOL statements for processing data—You can access a database by using SQL statements and then use COBOL statements to process and manipulate the data.
- System support for data consistency—If you require audited tables and views, the system maintains data consistency with the locking feature and the HP NonStop Transaction Management Facility (TMF) subsystem.

Development of a COBOL Program

You can embed static or dynamic SQL statements in a COBOL source file. You embed a static SQL statement as an actual SQL statement and run the SQL compiler to explicitly compile the statement before you run the program. To embed a dynamic SQL statement, code a placeholder variable for the statement, and then construct, SQL compile, and execute the statement at run time.

Host Variables

A host variable provides communication between COBOL statements and SQL statements. A host variable is a COBOL data item with a data type that corresponds to an SQL data type. You use host variables in SQL statements to receive data from a database or to insert data into a database.

You declare host variables in a Declare Section in the Data Division. A Declare Section is delimited by the BEGIN DECLARE SECTION and END DECLARE SECTION directives. In this example, FILENUMBER and MESSAGE are host variables:

```
DATA DIVISION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  FILENUMBER            PIC 9(6) DISPLAY.
01  MESSAGE               PIC X(200).
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The HP COBOL compiler accepts the CHARACTER SET clause in a host variable declaration to associate a single-byte or double-byte character set, such as Kanji, KSC5601, and ISO 8859/*n* with a host variable.

When you specify a host variable in an SQL statement, precede the host variable name with a colon (:). In COBOL statements, you do not need the colon as shown:

```
EXEC SQL
  SELECT COLUMN1 INTO :HOST-VARIABLE1 FROM =TABLEA
    WHERE COLUMN1 > :HOST-VARIABLE2
END-EXEC.
MOVE HOST-VARIABLE1 TO NEW-NAME.
...
```

For more information, see Section 2, Host Variables.

SQL/MP Statements and Directives

Table 1-1 on page 1-3 lists the SQL/MP statements and directives you can embed in a COBOL program.

Table 1-1. SQL/MP Statements and Directives

Type	Statement or Directive
Data Declaration	BEGIN DECLARE SECTION and END DECLARE SECTION
	INVOKE
	INCLUDE STRUCTURES
	INCLUDE SQLCA, INCLUDE SQLDA, and INCLUDE SQLSA
Data Definition Language (DDL)	ALTER CATALOG, ALTER COLLATION, ALTER INDEX, ALTER PROGRAM, ALTER TABLE, and ALTER VIEW
	COMMENT
	CREATE CATALOG, CREATE COLLATION, CREATE INDEX, CREATE PROGRAM, CREATE TABLE, and CREATE VIEW
	DROP
	HELP TEXT
	UPDATE STATISTICS
Data Manipulation Language (DML)	DECLARE CURSOR
	OPEN
	FETCH
	SELECT, INSERT, UPDATE, DELETE
	CLOSE
Data Status Language (DSL)	GET CATALOG OF SYSTEM
	GET VERSION (for SQL/MP software, catalogs, and objects)
	GET VERSION OF PROGRAM
Dynamic SQL Operations	PREPARE
	DESCRIBE and DESCRIBE INPUT
	EXECUTE and EXECUTE IMMEDIATE
	RELEASE
Error Processing	WHENEVER
Transaction Control	BEGIN WORK, COMMIT WORK, and ROLLBACK WORK

You code an SQL statement or directive by preceding it with EXEC SQL and then terminating it with END-EXEC. Example 1-1 shows static SQL statements embedded in a COBOL program:

Example 1-1. Static SQL Statements in a COBOL Program

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   IN-PARTS-REC.
     02   IN-PARTNUM    PIC 9(4) COMP.
     02   IN-PRICE      PIC S9(8)V99 COMP.
     02   IN-PARTDESC   PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
410-INSERT-DATA.
    MOVE 4120 TO IN-PARTNUM.
    MOVE 60000.00 TO IN-PRICE.
    MOVE "V8 DISK OPTION" TO IN-PARTDESC.
    EXEC SQL
      INSERT INTO SALES.PARTS
      (PARTNUM,     PRICE,      PARTDESC)
      VALUES (:IN-PARTNUM, :IN-PRICE, :IN-PARTDESC)
    END-EXEC.
```

For more information, see Section 3, SQL/MP Statements and Directives, and Section 4, Data Retrieval and Modification.

SQL/MP System Procedures

SQL/MP provides system procedures that perform various SQL operations and functions. For example, the SQLCA_DISPLAY2_ procedure returns error information from the SQLCA structure after an SQL statement executes. You call SQL system procedures from a COBOL program in the same manner you call other system procedures (for example, FILE_OPEN_, READ, WRITEREAD, and FILE_CLOSE_).

This example shows a call to the SQLCA_DISPLAY2_ procedure using all default parameters:

```
ENTER TAL "SQLCA_DISPLAY2_" USING SQLCA.
```

For more information, see Section 5, SQL/MP System Procedures, and Section 11, Character Processing Rules (CPRL) Procedures.

Program Compilation and Execution

The procedure to compile and execute an HP COBOL program that contains embedded SQL statements is similar to the steps you follow for an HP COBOL program that does not contain embedded SQL statements. You must perform only one extra step for an SQL program: you compile the embedded SQL statement by using the SQL compiler.

1. Add any required class MAP or class CATALOG DEFINEs

   Use class MAP DEFINEs to specify SQL objects—tables, views, indexes, and collations—and class CATALOG DEFINEs to specify SQL catalogs).

2. Run an HP COBOL compiler (COBOL85 or the NMCOBOL compiler), specifying a source file containing embedded SQL statements as input.

3. If necessary, run the Binder program (if you used the COBOL85 compiler), the `nld` or `ld` utility (if you used the native mode NMCOBOL compiler) to combine the COBOL object file with other object files.

4. Optionally, run the Accelerator on the COBOL object file to optimize it for execution on a TNS/R system.

5. Run the SQL compiler (SQLCOMP) to compile the SQL source statements in the COBOL object file and to validate the output SQL program file for execution.

6. Execute the SQL program file either interactively from TACL or the OSS prompt, or programmatically by using the COBOL CREATEPROCESS routine.

SQL/MP software supports the development of COBOL programs containing embedded SQL statements in both the Guardian and OSS environments. For more information, see Section 6, Explicit Program Compilation, and Section 7, Program Execution.

Error and Status Reporting

SQL/MP returns error and status information to a host-language program after the execution of each embedded SQL statement or directive. SQL/MP returns an SQL error or warning number to the SQLCODE variable and more extensive information to these SQL data structures:

- SQL communications area (SQLCA)—run-time information, including errors and warnings, generated by the most recently executed SQL statement.
- SQL statistics area (SQLSA)—statistics and performance information after the execution of DML statements and some dynamic SQL statements.
- SQL descriptor area (SQLDA)—information about input parameters and output variables in dynamic SQL statements.

For more information about the SQLCA and SQLSA structures, see Section 9, Error and Status Reporting. For information about the SQLDA structure, see Section 10, Dynamic SQL Operations.

Dynamic SQL Operations

With static SQL operations, you code the actual SQL statement in the COBOL source file. However, with dynamic SQL statements, a program can construct, compile, and execute an SQL statement at run time. You code a host variable as a placeholder for the dynamic SQL statement, which is usually unknown or incomplete until run time.

A dynamic SQL statement requires some input, often from a user at a terminal, to construct the final statement. The statement is constructed at run time from the user's input, compiled by the SQL compiler using a PREPARE statement, and then executed using an EXECUTE statement (or compiled and executed using an EXECUTE IMMEDIATE statement).

Example 1-2 shows a dynamic SQL operation that uses an INSERT statement similar to the static INSERT statement in Example 1-1 on page 1-4. In Example 1-1, the static INSERT statement is embedded in the source program code. In Example 1-2, the program dynamically builds the INSERT statement from information entered by a user.

Example 1-2. Dynamic SQL Statement in a COBOL Program

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01    INTEXT       PIC X(200).
...
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
  01 OPERATION    PIC X(200).
EXEC SQL END DECLARE SECTION  END-EXEC.
...
PROCEDURE DIVISION.
...
400-INSERT-DATA.
    MOVE INTEXT TO OPERATION.
    EXEC SQL EXECUTE IMMEDIATE :OPERATION  END-EXEC.
```

Example 1-2 accesses the PARTS table, which exists on a different subvolume. A user enters this information in the INTEXT variable to indicate the location of the PARTS table and other values needed to construct the INSERT statement:

```
INSERT INTO $VOL5.SALES.PARTS
   (PARTNUM, PRICE, PARTDESC)
   VALUES (4120, 60000.00, "V8 DISK OPTION")
```

The example builds the INSERT statement from information in the INTEX variable and moves the statement to the host variable named OPERATION. The host variable OPERATION is available to both COBOL and SQL statements. The example uses the EXECUTE IMMEDIATE statement to compile and execute the INSERT statement in OPERATION. (This example could also have used a PREPARE statement to compile the statement and an EXECUTE statement to execute it.)

For more information, see Section 10, Dynamic SQL Operations.

SQL/MP Version Management

Each product version update (PVU) of SQL/MP has an associated version number. The initial PVUs were version 1 (C10 and C20) and version 2 (C30). Version 300 of SQL/MP began using a three-digit version number to allow for software product revision (SPRs). A new version number is always greater than the previous number, but the new number might not follow a constant increment. For example, consecutive version numbers after version 315 might be 320, 325, and 340.

In addition, SQL objects (tables, indexes, views, collations, and constraints), programs, and catalogs have associated version numbers. This version number indicates the SQL features used by the SQL object or program and the SQL/MP software with which the SQL object or program is compatible. For example, a version 2 table might use the date-time data types or allow null values in a column. A version 2 table is compatible with SQL/MP software version 2 and 315 but is not compatible with version 1 software.

The version information in this manual includes these topics:

- Using compatible versions of the COBOL85 compiler, NMCOBOL compiler, SQL compiler, and SQL executor to compile and execute a program
- Using the data status language (DSL) statements: GET VERSION (for SQL objects, catalogs, and SQL/MP software), GET VERSION OF PROGRAM, and GET CATALOG OF SYSTEM
- Generating different versions of the SQLSA and SQLDA structures
- Converting a COBOL program written for version 1 or version 2 SQL/MP software to use version 300 (or later) SQL features and data structures
- Planning for future PVUs of SQL/MP

For additional information about version issues, see the *SQL/MP Version Management Guide*.

COBOL in the Open System (OSS) Environment

SQL/MP software supports the development of COBOL programs in the OSS environment as well as in the Guardian environment. In the OSS environment, you can code a COBOL program with a text editor such as `vi` and then use one of these HP COBOL compilers:

- `cobol` utility to invoke the COBOL85 compiler, Binder program, Accelerator, and SQL compiler. (The `cobol` utility is described in Section 6, Explicit Program Compilation.)
- `nmcobol` utility to invoke the NMCOBOL compiler, `ld` or `nld` linker, and SQL compiler on a TNS/R system

For more information, see Running the HP COBOL Compilers on page 6-12. Most features of the COBOL language and library are available in the OSS environment, and most of them operate as they do in the Guardian environment. Differences in the two environments as they relate to the COBOL interface to SQL/MP are discussed throughout this manual. The *COBOL85 for NonStop Systems Manual* contains detailed information on using COBOL in the OSS environment.

Effect on Conformance to ISO/ANSI Standards

When an HP COBOL program does not use the SQL directive (which notifies the COBOL85 or NMCOBOL compiler to expect embedded SQL), embedded SQL does not affect HP COBOL conformance to the COBOL ISO/ANSI standard. When a program does use the SQL directive, embedded SQL affects HP COBOL conformance to the COBOL ISO/ANSI standard only to the extent required to process SQL statements. For information on HP COBOL conformance to the COBOL ISO/ANSI standard, see the *COBOL85 for NonStop Systems Manual*.

The HP COBOL embedded SQL implementation conforms to the ANSI Database—Embedded SQL Standard (ANSI X3.168-1989), with the restrictions and extensions mentioned in this section.

2 Host Variables

A host variable is a data item you can use in both COBOL and NonStop SQL/MP statements to provide communication between these two types of statements. A host variable appears as a COBOL name and can be any COBOL data item declared in a Declare Section that has a corresponding SQL data type as shown in Table 2-1, Corresponding SQL and COBOL Data Types, on page 2-3.

For static SQL operations, a host variable can be an input or output variable (or both in some cases) in SQL statements. An input variable transfers data from the program to the database, whereas an output variable transfers data from the database to the program. (For dynamic SQL operations, input parameters and output variables fulfill the same function as input and output host variables in static SQL statements.)

An indicator variable is a two-byte integer variable, also declared in the Declare Section, that is associated with a host variable. An indicator variable indicates whether a column contains, or can contain, a null value. A null value means that a value is either unknown for the row or does not apply to the row. A program uses an indicator variable to insert null values into a database or to test a column value for a null value after retrieving the value from a database.

Topics include:

Specifying a Declare Section

You declare all host variables in a Declare Section. The BEGIN DECLARE SECTION and END DECLARE SECTION directives designate a Declare Section. Follow these guidelines when you specify a Declare Section:

- Use the BEGIN DECLARE SECTION and END DECLARE SECTION directives only in pairs. A period after the END-EXEC key words for either directive is ignored.

- Place a Declare Section in the Data Division. You can specify more than one Declare Section in a program, if necessary, but you cannot nest Declare Sections.
- Do not place a Declare Section within a COBOL record description.
- The first item after the BEGIN DECLARE SECTION directive must have level 01.
- The only directives you can specify in a Declare Section are the COBOL compiler SOURCE directive and the SQL INVOKE directive.
- Use COBOL comment statements to document a Declare Section.

This example shows the declaration of host variables in a Declare Section:

```
DATA DIVISION.
..
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 CUSTNUM          PIC S9(4) COMP.
01 CITY             PIC X(14).
EXEC SQL INVOKE SALES.PARTS AS SALES-REC END-EXEC.
?SOURCE COBLIB(DECLARES)
..
EXEC SQL END DECLARE SECTION END-EXEC.
```

Coding Host Variable Names

Use COBOL naming conventions for host variable names. A COBOL name can contain from 1 to 30 alphanumeric characters, including letters, digits, and hyphens (-). The first or last letter cannot be a hyphen. Letters can be uppercase, lowercase, or a combination of both. HP COBOL names must contain at least one letter or hyphen. You must also avoid using names that conflict with these SQL structures:

- SQLINALL internal structure
- SQLCA, SQLSA, and SQLDA structures

To use a COBOL record description as a host variable, specify the record name as a level 01 entry and use level numbers 01 to 49, 66, 77, and 88 for the host variables. The individual data items, and not the record name, are the host variables. You must use declarations compatible with the SQL data types as shown in Table 2-1 on page 2-3. You must also observe certain restrictions for the PICTURE clause. For more information, see Using the COBOL PICTURE Clause on page 2-7.

Using Corresponding SQL and COBOL Data Types

Table 2-1 on page 2-3 lists the corresponding SQL and COBOL data types. You can specify a COBOL data item as a host variable, if the COBOL data item has a corresponding SQL data type.

Table 2-1. Corresponding SQL and COBOL Data Types (page 1 of 2)

SQL/MP Data Type	COBOL Data Type
Fixed-Length Character Data Type	
CHARACTER (*l*) PIC X(*l*).	PIC X(*l*)
Fixed-Length Character Data Type With CHARACTER SET Clause	

CHARACTER (*l*)
  CHARACTER SET *charset*
PIC X(*l*)
  CHARACTER SET *charset*
	01 *column-name*
	   CHARACTER SET *charset*
	   PIC X(*l*).

Fixed-Length Character Data Type With NATIONAL CHARACTER Clause	

NATIONAL CHARACTER (*l*)	01 *column-name*
	   CHARACTER SET *def-charset*
	   PIC X(*l*).

Variable-Length Character Data Type	

VARCHAR(*l*)	02 *column-name*.
	   03 LEN PIC S9(4) COMP.
	   03 VAL PIC X(*l*).

Variable-Length Character Data Type With CHARACTER SET Clause	

VARCHAR(*l*) @@@
  CHARACTER SET *charset*
	01 *column-name*.
	   02 LEN PIC S9(4) COMP.
	   02 VAL CHARACTER SET *charset*
	        PIC X(*l*).

Variable-Length Character Data Type With NATIONAL CHARACTER Clause	

NATIONAL CHARACTER
  VARYING(*l*)
	01 *column-name*.
	   02 LEN PIC S9(4) COMP.
	   02 VAL CHARACTER SET *def-charset*
	        PIC X(*l*).

Numeric Data Types	
NUMERIC (1 to 4,*s*) SIGNED	PIC S9(4-*s*)V9(*s*) COMP.
NUMERIC (1 to 4,*s*) UNSIGNED	PIC 9(4-*s*)V9(*s*) COMP.
NUMERIC (5 to 9,*s*)SIGNED	PIC S9(9-*s*)V9(*s*) COMP.

l is a positive integer that represents the length in characters.
charset is one of these character-set keywords: KANJI, KSC5601, ISO8859*n*, where *n* is 1 – 9, or UNKNOWN (a single-byte unknown character set). *charset* must be enclosed in double quotation marks (").
def-charset is the default multibyte character set. *def-charset* is KANJI, unless it is otherwise changed or set during system generation.
s is a positive integer that represents the scale of the number.
HP COBOL treats BINARY as COMPUTATIONAL (or COMP). Therefore, references to COMPUTATIONAL (or COMP) also apply to BINARY.

The INTERVAL data type has an extra byte to store a sign. This extra byte can contain a blank, plus, or minus.
Indicator variables have the SQL data type SMALLINT SIGNED and the COBOL data type PIC S9(4) COMP.

Table 2-1. Corresponding SQL and COBOL Data Types (page 2 of 2)

SQL/MP Data Type	COBOL Data Type
`NUMERIC (5 to 9,`*`s`*`) UNSIGNED`	`PIC  9(9-`*`s`*`)V9(`*`s`*`) COMP.`
`NUMERIC (10 to 18,`*`s`*`) SIGNED`	`PIC S9(18-`*`s`*`)V9(`*`s`*`) COMP.`
`DECIMAL (`*`l`*`,`*`s`*`) SIGN IS LEADING`	`PIC S9(`*`l`*`-`*`s`*`)V9(`*`s`*`) DISPLAY SIGN IS LEADING.`
`DECIMAL (`*`l`*`,`*`s`*`) UNSIGNED`	`PIC 9(`*`l`*`-`*`s`*`)V9(`*`s`*`) DISPLAY.`
`PIC 9(`*`l`*`-`*`s`*`)V9(`*`s`*`) COMP`	`Same as NUMERIC.`
Numeric Data Types	
`PIC 9(`*`l`*`-`*`s`*`)V9(`*`s`*`)`	`Same as DECIMAL.`
`SMALLINT SIGNED`	`PIC S9(4) COMP.`
`SMALLINT UNSIGNED`	`PIC  9(4) COMP.`
`INTEGER SIGNED`	`PIC S9(9) COMP.`
`INTEGER UNSIGNED`	`PIC  9(9) COMP.`
`LARGEINT SIGNED`	`PIC S9(18) COMP.`
`FLOAT (1 to 22 bits)`	`Not supported.`
`REAL`	`Not supported.`
`FLOAT (23 to 54 bits)`	`Not supported.`
`DOUBLE PRECISION`	`Not supported.`
Date-Time and INTERVAL Data Types	
`DATETIME`	`PIC X(26).`
`DATE`	`PIC X(10).`
`TIME`	`PIC X(8).`
`TIMESTAMP`	`PIC X(26).`

`l` is a positive integer that represents the length in characters.
`charset` is one of these character-set keywords: KANJI, KSC5601, ISO8859*`n`*, where *`n`* is 1 – 9, or UNKNOWN (a single-byte unknown character set). *`charset`* must be enclosed in double quotation marks (").
`def-charset` is the default multibyte character set. *`def-charset`* is KANJI, unless it is otherwise changed or set during system generation.
`s` is a positive integer that represents the scale of the number.
HP COBOL treats BINARY as COMPUTATIONAL (or COMP). Therefore, references to COMPUTATIONAL (or COMP) also apply to BINARY.

The INTERVAL data type has an extra byte to store a sign. This extra byte can contain a blank, plus, or minus.
Indicator variables have the SQL data type SMALLINT SIGNED and the COBOL data type PIC S9(4) COMP.

Note. To retrieve floating-point columns, you must declare all the required host variables with corresponding data types supported by COBOL. (Floating-point columns will be handled during data type conversion.) For example, you must declare numeric host variables, like REAL, FLOAT, and DOUBLE PRECISION, as a numeric data type supported by COBOL.

Ensure that the actual values in a floating-point column can be converted without resulting in an overflow.

Data Conversion

SQL/MP performs the conversion between SQL and COBOL data types:

- When a host variable serves as an input variable (supplies a value to the database), SQL/MP first converts the value that the variable contains to a compatible SQL data type and then uses the value in the SQL operation.
- When a host variable serves as an output variable (receives a value from a database), SQL/MP converts the value to the data type of the host variable.

Note. For systems running J06.09 and later J-series RVUs, H06.20 and later RVUs, or G06.32 and later G-series RVUs, the SQL integer data types is mapped to the corresponding COBOL data types with the `COMP-5` option when the external DEFINE `=_SQL_MAPTO_COBOL_COMP5` is present in the system.

SQL/MP supports conversion within character types and numeric types, not between character and numeric types.

For conversion between character strings of different lengths, SQL/MP pads the receiving string on the right with blanks as necessary. If the receiving string is too small, SQL/MP truncates the right part of the longer string and generates a warning code in the SQLCODE variable.

If an input value is too large for the SQL column, SQL/MP reports error 8300 (file-system error encountered). If you are using the SQLCA_DISPLAY2_ procedure to display error messages, the specific file-system error (1031) is also returned.

For numeric type conversion, SQL/MP converts data between signed and unsigned types and between types with different precisions. Scales and precisions can be different. Decimal variables can have different sign placements.

Note. For optimal performance, declare host variables with the same data types and lengths as their respective columns in SQL statements. This programming practice minimizes the data conversions performed by SQL/MP and, therefore, can improve the performance of your program.

CAST Function

The CAST function allows you to convert a parameter from one data type to another data type (character and numeric data types only) in dynamic SQL statements. For information about the CAST function, see the *SQL/MP Reference Manual*.

Specifying Host Variables in SQL Statements

Use this syntax to specify a host variable in an SQL statement. You must precede the host variable name with a colon (:). The colon causes the HP COBOL compiler to handle the name as a host variable.

```
:host-variable [ { OF | IN } record-name ]

 [[ INDICATOR ]:indicator-host-variable [ OF record-name ] ]

 [ TYPE AS { DATETIME [start-date-time TO] end-date-time}  ]
 [         {                                            }  ]
 [         { DATE                                       }  ]
 [         {                                            }  ]
 [         { TIME                                       }  ]
 [         {                                            }  ]
 [         { TIMESTAMP                                  }  ]
 [         {                                            }  ]
 [         { INTERVAL start-date-time                   }  ]
 [         {    [ ( start-field-precision ) ]           }  ]
 [         {    [ TO end-date-time ]                    }  ]
```

`host-variable [ { OF | IN } record-name ]`

specifies the name of the host variable as declared in the program. `record-name` specifies a level 01 item. The host variable name must be qualified by the record name or group item name only if the data item name is not unique in the program.

`INDICATOR indicator-host-variable [ OF record-name ]`

specifies an indicator variable to handle null values that might be returned to the host variable or to insert null values into the database through the host variable. If you omit the keyword INDICATOR before the variable name, an indicator is assumed because host variable names are separated by commas.

Declare `indicator-host-variable` as a data item of type PIC S9(4) COMP. For a value returned to the host variable from the database, SQL/MP sets the indicator variable to -1 if the value is null or 0 if the value is not null.

To insert null values into a database, set the indicator variable to a value less than 0 for a null value or a value equal to or greater than 0 for a nonnull a value.

For more information about indicator values, see Using Indicator Variables for Null Values on page 2-11.

`TYPE AS`

specifies that the host variable will have the specified date-time or INTERVAL data type. A host variable that is to contain date or time values must be defined with a character data type.

Using the COBOL PICTURE Clause

If you use the PICTURE clause to declare COBOL record descriptions as host variables, the clause must conform to both COBOL syntax rules and SQL/MP limitations.

Fixed-Length Character Data

Use the PICTURE clause to declare a host variable for fixed-length character data (CHAR data type):

```
PICTURE X (length) [ USAGE IS DISPLAY ]
```

The *length* value must be a positive integer and not greater than 4096. Instead of *length*, you can specify multiple Xs, with each X representing one character position, or you can specify multiple Xs and lengths as allowed in COBOL. For example, PIC XXX(3)X(3) is valid. DISPLAY is the default.

Variable-Length Character Data

Use a group item with two data items to declare a host variable for variable-length character data (VARCHAR data type):

```
nn group-name.
     nm LEN      PIC S9(4) COMP.
     nm VAL      PIC X(len).
```

The *group-name* must follow COBOL naming conventions. The level numbers are indicated by *nn* and *nm*: *nn* can be any level in the range 01 to 49, and *nm* is a greater level than *nn*. LEN specifies the actual length of the character item in VAL. VAL is a character data item with *len* specifying the maximum number of characters that can be stored in VAL.

For example, the EMPLOYEE table has the EMP-NAME column defined as VARCHAR(18). In a COBOL program, the column definition is:

```
05  EMP-NAME.
    10  LEN  PIC S9(4) COMP.
    10  VAL  PIC X(18).
```

In the Procedure Division, you must explicitly move a value to LEN before using EMP-NAME in an SQL statement:

```
MOVE "SMITH" TO VAL OF EMP-NAME.
MOVE 5 to LEN OF EMP-NAME.
EXEC SQL INSERT INTO EMPLOYEE-TABLE(EMP_NAME)
              VALUES (:EMP-NAME)
END-EXEC.
```

Numeric Data

Use the PICTURE clause to declare a host variable for numeric data (NUMERIC, DECIMAL, SMALLINT, LARGEINT, and INTEGER data types):

```
PICTURE [S] { 9(integer) [ V [ 9(scale) ] ] }
            { V9(scale)                     }

  [ [ USAGE IS ]  { DISPLAY        } ]
  [               { COMPUTATIONAL  } ]
  [               { COMP           } ]
  [               { BINARY         } ]
```

If you specify COMPUTATIONAL, COMP, or BINARY, the value is stored as a binary integer with an implied decimal point. If you omit the USAGE clause, DISPLAY is the default, and the digits are stored as ASCII characters.

The S specifies a signed variable. If you omit S, the variable is unsigned. The `9(integer)` specifies *`integer`* number of digits; *`integer`* must be positive. The V designates a decimal position. The `9(`*`scale`*`)` designates the number of positions to the right of the decimal. The value of *`scale`* must be a positive integer. If you do not specify *`scale`*, the value 0 is used.

Instead of *`integer`* or *`scale`*, you can specify multiple 9s, with each 9 representing one digit. You can also specify multiple 9s, integers, or scales as allowed in COBOL. For example, PIC 9V9 has a scale of 1. PIC 999(4)V999 is equivalent to PIC 9(6)V9(3) and has a scale of 3.

The values of *`integer`* and *`scale`* determine the size of the column. The sum of these values cannot exceed 18.

There is no default numeric column definition. You must specify either `9(`*`integer`*`)` or `V9(`*`scale`*`)`.

You must ensure that the value limit imposed by the PICTURE clause of COMP items is valid for the data. Corresponding SQL columns defined as type NUMERIC, SMALLINT, INTEGER, LARGEINT, or with COMPUTATIONAL can accept values as large as the limit determined by the column size in bytes. For example, the COBOL item described as PIC S9(4) COMP corresponds to an SQL integer column. SQL/MP allows five-digit values up to 32767, but COBOL allows only four digits (maximum value 9999).

Note. For systems running J06.09 and later J-series RVUs, H06.20 and later RVUs, or G06.32 and later G-series RVUs, the SQL integer data types limitations are overridden when the external `DEFINE =_SQL_MAPTO_COBOL_COMP5` is present in the system. In these cases, the SQL integer columns are mapped to the corresponding COBOL data types with `COMP-5` option which rules out these value limitations

Using COBOL Data Description Clauses

The next table summarizes the COBOL data description clauses and their interpretation by SQL/MP when they are used in host variable declarations. SQL/MP does not support the COBOL special names option DECIMAL POINT IS COMMA.

Using Date-Time and INTERVAL Data Types

COBOL Description	SQL/MP Host Variable Interpretation
BLANK	The clause is ignored.
data-name	Any data name is allowed, including an SQL reserved word. Specific hyphenation rules apply.
FILLER	The clause is ignored.
JUSTIFIED	The clause is not allowed. However, it can appear in an entry already being ignored, such as REDEFINES.
level number	Any number is allowed. Entries with the level number 66 or 88 are ignored.
OCCURS	The clause is not allowed. However, it can appear in an entry already being ignored, such as REDEFINES.
PICTURE	The clause must be consistent with the PICTURE clause rules for host variables.
REDEFINES	The clause is ignored.
SIGN	No restrictions apply, and the appropriate conversion for SQL data types is made.
SYNC	The clause is ignored.
USAGE	The USAGE options correspond to these SQL data types: ● COMPUTATIONAL (COMP) or BINARY to SQL type NUMERIC or to an integer type (SMALLINT, INTEGER, or LARGEINT). ● DISPLAY to character (for PIC X) or decimal (for PIC 9). ● The INDEX and PACKED-DECIMAL options are not allowed.
VALUE	The clause is ignored.

The SQL date-time and INTERVAL data types that you can use in host variable declarations are:

Data Type	Description
DATETIME	Represents a date and time from year to microsecond (logical subsets, such as MONTH TO DAY, are allowed)
DATE	Represents a date and is equivalent to DATETIME YEAR TO DAY

TIME	Represents a time and is equivalent to DATETIME HOUR TO SECOND
TIMESTAMP	Represents a date and time and is equivalent to DATETIME YEAR TO FRACTION(6)
INTERVAL	Represents a duration of time as a year-month or day-time interval

Declare date-time values as character data types and then use the TYPE AS clause to direct SQL/MP to interpret the value in the host variable as a date-time or INTERVAL value. Sample TYPE AS clauses are:

- TYPE AS DATETIME YEAR TO HOUR
- TYPE AS DATE
- TYPE AS TIME
- TYPE AS TIMESTAMP
- TYPE AS INTERVAL YEAR

You can insert or retrieve date-time values in any of three formats, independently of the SQL column definition. For example, you can specify formats such as 06/15/1996, 1996-06-15, or 15.06.1996. You must declare the host variable size to be consistent with the format you will use. You then control the display format by retrieving the value by using the DATEFORMAT function.

Using Indicator Variables for Null Values

A null value in an SQL column indicates that a value is either unknown for the row or is not applicable to the row. If a column allows null values, a program can use an indicator variable to set or receive the column value. An indicator variable is a two-byte integer variable, defined in the Declare Section, associated with the host variable that sets or receives the actual column value.

The INVOKE directive automatically declares indicator variables for columns defined to allow null values. For information, see Using Indicator Variables With the INVOKE Directive on page 2-22.

To send a value to SQL/MP for insertion, update, or comparison, a program sets the indicator variable to less than zero (0) for a null value or zero (0) for a nonnull value. When it returns a value that allows a null value to a program, SQL/MP sets the indicator variable to less than zero (0) for a null value or zero (0) for a nonnull value.

A program can use indicator variables to perform these operations:

- To insert null values into a database with an INSERT or UPDATE statement
- To test for a null value after retrieving a value from a database with a SELECT statement

Inserting a Null Value

To insert a null value, the program sets the indicator variable to a value less than 0 for the column receiving the null value before executing the INSERT statement. This INSERT statement uses an indicator variable to insert a null value into the RETIREES table:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 RETIREE-REC.
   02 EMPNUM       PIC 9(5) COMP.
   02 RETIRE-DATE  PIC X(10).
   02 RETIRE-IND   PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION  END-EXEC.
...
PROCEDURE DIVISION.
...
MOVE NULL-EMPNUM TO EMPNUM.
MOVE -1 TO RETIRE-IND.

EXEC SQL
  INSERT INTO =RETIREES
  VALUES (:EMPNUM,:RETIRE-DATE INDICATOR :RETIRE-IND)
END-EXEC.
...
```

The next example uses the NULL keyword instead of an indicator variable to insert the null value:

```
MOVE NULL-EMPNUM TO EMPNUM.
...
EXEC SQL
  INSERT INTO =RETIREES VALUES (:EMPNUM, NULL)
END-EXEC.
```

Testing for a Null Value

To test for a null value, a program tests the indicator variable associated with a host variable. This example selects data from the PRODUCTS table and then tests for a null value using the indicator variable SHIP-IND. After the SELECT statement executes, the example tests the indicator variable for a null value. If the value of the indicator variable is less than 0, the associated column contains a null value.

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 PRODUCT-REC.
    02 PRODNUM        PIC 9(5) COMP.
    02 DATE-SHIPPED   PIC X(10).
    02 SHIP-IND       PIC 9(4) COMP.
 ...
 EXEC SQL END DECLARE SECTION END-EXEC.

* Variable for displaying the date or NULL:
  01 VALUE-DISPLAY  PIC X(10) VALUE SPACES.
...
* Declare a cursor to perform the SELECT:
 EXEC SQL DECLARE GET-PRODNUM CURSOR FOR
   SELECT PRODNUM, DATE-SHIPPED FROM =PRODUCTS
   WHERE PRODNUM > MAX-PRODNUM
 END-EXEC.

 PROCEDURE DIVISION.
 0100-MAIN.
 ...
  EXEC SQL OPEN GET-PRODNUM END-EXEC.
  PERFORM 0150-SELECT UNTIL SQLCODE OF SQLCA NOT = 0.
  EXEC SQL CLOSE GET-PRODNUM END-EXEC.
 ...
 0150-SELECT.
  EXEC SQL FETCH GET-PRODNUM INTO
    :PRODNUM, :DATE-SHIPPED INDICATOR :SHIP-IND
  END-EXEC.
* NonStop SQL/MP sets SHIP-IND to -1 if the column
* contained a null value in the selected row.

 IF SHIP-IND = -1 THEN MOVE "NULL" TO VALUE-DISPLAY
    ELSE MOVE DATE-SHIPPED TO VALUE-DISPLAY.
 IF SQLCODE = 0  DISPLAY PRODNUM "   " VALUE-DISPLAY.
 ...
```

Retrieving Rows With Null Values

You use an indicator variable to insert null values into a database or to test for a null value after you retrieve a row. However, you cannot use an indicator variable set to -1 in a WHERE clause to retrieve a row that contains a null value. If you use an indicator variable set to -1 in a WHERE clause, SQL/MP does not find the row and returns an SQLCODE of 100, even if a column actually contains a null value.

To retrieve a row that contains a null value, use the NULL predicate in the WHERE clause. For example, to retrieve rows that have null values from the EMPLOYEE table using a cursor, specify the NULL predicate in the WHERE clause in the associated SELECT statement when you declare the cursor:

```
* Declare a cursor to find rows with null salaries.
 EXEC SQL DECLARE GET-NULL-SALARY CURSOR FOR
   SELECT EMPNUM, FIRST-NAME, LAST-NAME,
          DEPTNUM, JOBCODE, SALARY
   FROM =EMPLOYEE
   WHERE SALARY IS NULL
 END-EXEC.
 ...
 PROCEDURE DIVISION.
 100-MAIN.
 ...
 EXEC SQL OPEN GET-NULL-SALARY END-EXEC.
 PERFORM 200-FETCH-NULL UNTIL SQLCODE OF SQLCA = 100.
 EXEC SQL CLOSE GET-NULL-SALARY END-EXEC.
 ...
 200-FETCH-NULL.

 EXEC SQL FETCH GET-NULL-SALARY INTO
   :EMPNUM OF EMPLOYEE-RECORD,
   :FIRST-NAME OF EMPLOYEE-RECORD
   :LAST-NAME OF EMPLOYEE-RECORD
   :DEPTNUM OF EMPLOYEE-RECORD
   :JOBCODE OF EMPLOYEE-RECORD
   :SALARY OF EMPLOYEE-RECORD
 END-EXEC.

* Process the row that contains the null salary.
 ...
```

Creating Host Variables Using the INVOKE Directive

The INVOKE directive creates host variables that correspond to columns in an SQL table or view. Each host variable is a COBOL data item with the same name as the respective column in the table or view. If a column allows null values, INVOKE also generates an indicator variable for the column.

To execute an INVOKE directive, a process started by the program must have read access to the invoked tables or views during COBOL compilation. For information about process access, see Required Access Authority on page 7-1.

You code the INVOKE directive in a Declare Section. The HP COBOL compiler checks the host variables generated by INVOKE for naming conflicts with other host variables in the Declare Section, but not with COBOL reserved words. (If a name conflict occurs, use SQLCI to generate the record description in a file, edit the names that conflict with the reserved words, and copy the modified record description into your source program. For more information, see Using INVOKE With SQLCI on page 2-24.)

Advantages of Using an INVOKE Directive

You can code host variables by creating a COBOL record definition that corresponds to the SQL table. However, using an INVOKE directive to generate host variables has these advantages:

- Program independence—If you modify a table or view, the INVOKE directive recreates the host variables to correspond to the new table or view when you recompile the program. (You must, however, modify a program that refers to a deleted column or must access a new column.)
- TACL DEFINEs—The INVOKE directive accepts a class MAP DEFINE name for a table or view name (but not for a record name).
- Program performance—The INVOKE directive maps the SQL data types to the corresponding COBOL data types. No data conversion is required at run time.
- Program readability and maintenance—The INVOKE directive creates host variables using the same names as column names in the table or view and generates comments that show the table or view name and the time and date of the definition.

COBOL Record Descriptions

The next examples show the correspondence between columns of various SQL data types and the COBOL record description generated by the INVOKE directive. Example 2-1 shows the CREATE TABLE statements that generate the SQL tables.

Example 2-1. CREATE TABLE Statements (page 1 of 2)

```
CREATE TABLE \NEWYORK.$DISK1.SQL.TYPECOB2 (
TYPE_CHAR1           CHARACTER (10) CHARACTER SET ISO88591 NOT NULL,
TYPE_CHAR1_NULL      CHARACTER (10) CHARACTER SET ISO88591         ,
TYPE_CHAR2           CHARACTER (10) CHARACTER SET KANJI    NOT NULL,
TYPE_CHAR2_NULL      CHARACTER (10) CHARACTER SET KANJI            ,
TYPE_VARCHAR1        VARCHAR   (10) CHARACTER SET ISO88591 NOT NULL,
TYPE_VARCHAR1_NULL   VARCHAR   (10) CHARACTER SET ISO88591         ,
TYPE_VARCHAR2        VARCHAR   (10) CHARACTER SET KANJI    NOT NULL,
TYPE_VARCHAR2_NULL   VARCHAR   (10) CHARACTER SET KANJI            ,
TYPE_NCHAR_F         NATIONAL CHARACTER (10)               NOT NULL,
TYPE_NCHAR_F_NULL    NATIONAL CHARACTER (10)                       ,
TYPE_NCHAR_V         NATIONAL CHARACTER VARYING (10)       NOT NULL,
TYPE_NCHAR_V_NULL    NATIONAL CHARACTER VARYING (10)               ,
TYPE_COB_PICX1       PIC X(10) CHARACTER SET ISO88591      NOT NULL,
TYPE_COB_PICX1_NULL  PIC X(10) CHARACTER SET ISO88591              ,
TYPE_COB_PICX2       PIC X(10) CHARACTER SET KANJI         NOT NULL,
TYPE_COB_PICX2_NULL  PIC X(10) CHARACTER SET KANJI
)  CATALOG  $SQL.SQLCAT ;
CREATE TABLE \NEWYORK.$DISK1.SQL.TYPECOB2 (
TYPE_CHAR1           CHARACTER (10) CHARACTER SET ISO88591 NOT NULL,
TYPE_CHAR1_NULL      CHARACTER (10) CHARACTER SET ISO88591         ,
TYPE_CHAR2           CHARACTER (10) CHARACTER SET KANJI    NOT NULL,
TYPE_CHAR2_NULL      CHARACTER (10) CHARACTER SET KANJI            ,
TYPE_VARCHAR1        VARCHAR   (10) CHARACTER SET ISO88591 NOT NULL,
TYPE_VARCHAR1_NULL   VARCHAR   (10) CHARACTER SET ISO88591         ,
TYPE_VARCHAR2        VARCHAR   (10) CHARACTER SET KANJI    NOT NULL,
TYPE_VARCHAR2_NULL   VARCHAR   (10) CHARACTER SET KANJI            ,
TYPE_NCHAR_F         NATIONAL CHARACTER (10)               NOT NULL,
TYPE_NCHAR_F_NULL    NATIONAL CHARACTER (10)                       ,
TYPE_NCHAR_V         NATIONAL CHARACTER VARYING (10)       NOT NULL,
TYPE_NCHAR_V_NULL    NATIONAL CHARACTER VARYING (10)               ,
TYPE_COB_PICX1       PIC X(10) CHARACTER SET ISO88591      NOT NULL,
```

Example 2-1. CREATE TABLE Statements (page 2 of 2)

```
TYPE_COB_PICX1_NULL PIC X(10) CHARACTER SET ISO88591              ,

TYPE_COB_PICX2      PIC X(10) CHARACTER SET KANJI         NOT NULL,

TYPE_COB_PICX2_NULL PIC X(10) CHARACTER SET KANJI
)  CATALOG  $SQL.SQLCAT ;
```

These INVOKE directives are coded in a COBOL source file:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL
  INVOKE \NEWYORK.$DISK1.SQL.TYPECOB1 AS TYPES-REC1 END-EXEC.

EXEC SQL
  INVOKE \NEWYORK.$DISK1.SQL.TYPECOB2 AS TYPES-REC2 END-EXEC.
EXEC SQL END DECLARE SECTION;
```

Example 2-2 shows the record descriptions generated by the INVOKE directives.

Example 2-2. COBOL Record Descriptions Generated by the INVOKE Directive (page 1 of 2)

```
*  Record Definition for table \NEWYORK.$DISK1.SQL.TYPECOB1
*  Definition current at 15:55:34 - 10/10/94
 01 TYPES-REC1.
    02 TYPE-CHAR                PIC X(10).
    02 TYPE-VARCHAR.
       03 LEN                   PIC S9(4) COMP.
       03 VAL                   PIC X(10).
    02 TYPE-NUM4-S              PIC S9(4) COMP.
    02 TYPE-NUM4-U             PIC 9(4) COMP.
    02 TYPE-NUM9-S              PIC S9(7)V9(2) COMP.
    02 TYPE-NUM9-U              PIC 9(7)V9(2) COMP.
    02 TYPE-NUM18-S             PIC S9(16)V9(2) COMP.
    02 TYPE-SMALLINT-S          PIC S9(4) COMP.
    02 TYPE-SMALLINT-U          PIC 9(4) COMP.
    02 TYPE-INT-S               PIC S9(9) COMP.
    02 TYPE-INT-U               PIC 9(9) COMP.
    02 TYPE-LARGEINT-S          PIC S9(18) COMP.
*  TYPE-FLOAT: DOUBLE PRECISION IS NOT SUPPORTED
*  TYPE-REAL: REAL IS NOT SUPPORTED
*  TYPE-DOUBLE-PREC: DOUBLE PRECISION IS NOT SUPPORTED
    02 TYPE-DEC-S               PIC S9(16)V9(2) DISPLAY SIGN IS LEADING.
    02 TYPE-DEC-U               PIC 9(7)V9(2) DISPLAY.
    02 TYPE-COB-PIC9            PIC 9(9) COMP.
    02 TYPE-COB-PICX            PIC X(10).
    02 TYPE-DATETIME            PIC X(26).
    02 TYPE-DATE                PIC X(10).
    02 TYPE-TIME                PIC X(8).
    02 TYPE-INTERVAL            PIC X(6).
    02 TYPE-ZCHAR-NULL-OK-I     PIC S9(4) COMP.
    02 TYPE-ZCHAR-NULL-OK       PIC X(10).
    02 TYPE-ZNUM-NULL-OK-I      PIC S9(4) COMP.
    02 TYPE-ZNUM-NULL-OK        PIC S9(4) COMP.
```

Example 2-2. COBOL Record Descriptions Generated by the INVOKE Directive (page 2 of 2)

```
*  Record Definition for table \NEWYORK.$DISK1.SQL..TYPECOB2
*  Definition current at 15:55:38 - 10/10/94
 01 TYPES-REC2.
    02 TYPE-CHAR1              CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-CHAR1-NULL-I       PIC S9(4) COMP.
    02 TYPE-CHAR1-NULL         CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-CHAR2              CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-CHAR2-NULL-I       PIC S9(4) COMP.
    02 TYPE-CHAR2-NULL         CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-VARCHAR1.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-VARCHAR1-NULL-I    PIC S9(4) COMP.
    02 TYPE-VARCHAR1-NULL.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-VARCHAR2.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-VARCHAR2-NULL-I    PIC S9(4) COMP.
    02 TYPE-VARCHAR2-NULL.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-NCHAR-F            CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-NCHAR-F-NULL-I     PIC S9(4) COMP.
    02 TYPE-NCHAR-F-NULL       CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-NCHAR-V.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-NCHAR-V-NULL-I     PIC S9(4) COMP.
    02 TYPE-NCHAR-V-NULL.
       03 LEN                  PIC S9(4) COMP.
       03 VAL                  CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-COB-PICX1          CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-COB-PICX1-NULL-I   PIC S9(4) COMP.
    02 TYPE-COB-PICX1-NULL     CHARACTER SET "ISO88591" PIC X(10).
    02 TYPE-COB-PICX2          CHARACTER SET "KANJI" PIC X(10).
    02 TYPE-COB-PICX2-NULL-I   PIC S9(4) COMP.
    02 TYPE-COB-PICX2-NULL     CHARACTER SET "KANJI" PIC X(10).
```

When you use the INVOKE directive to generate host variables, the HP COBOL compiler writes a COBOL data description for each column in the specified table or view. In some cases, however, the compiler must convert an SQL column name or data type as described:

Column or Data Type	**Description of Change**
Underscore (_) within a name	Converts underscores to hyphens (–). For example, the column name CITY_STREET becomes CITY-STREET.
Underscore (_) at the end of a name	Truncates the underscore so that the resulting COBOL name does not end in a hyphen. For example, the column name HOME_ becomes HOME.

Column with VARCHAR data type	Creates a group item with two elementary data items. The group item name is derived from the VARCHAR column name. The data names of the subordinate data items are:
	● LEN, a numeric data item for the length.
	● VAL, a fixed-length character data item for the string, with the maximum length specified by the VARCHAR column definition.
	For example, CUSTNAME defined as VARCHAR (26) becomes this group item:
	`02 CUSTNAME.`
	`   03 LEN      PIC S9(4) COMP.`
	`   03 VAL      PIC X(26).`

DATETIME, DATE, TIME, TIMESTAMP, or INTERVAL data type	Converts columns to character fields. The size is determined by the date-time or INTERVAL fields. INTERVAL columns have an additional byte for a sign (that is, a negative interval is possible). The format of the column is the DEFAULT format (ANSI).

Embedded Sign in a Decimal Data Type

SQL/MP supports only leading embedded signs for columns defined with a decimal data type. Therefore, you cannot directly specify or use INVOKE to generate host variables with embedded trailing signs from numeric columns defined as either of these:

`PICTURE S 9(`*`n`*`) DISPLAY` or `DECIMAL (`*`n`*`,`*`s`*`) SIGNED`

To use a trailing sign with a numeric variable, follow these steps:

1. Use INVOKE to declare a host variable with a leading embedded sign.
2. Define a corresponding COBOL variable with a trailing sign.
3. Read data into the host variable with a leading sign.
4. Move the data to the COBOL variable with a trailing sign.

System-Defined Primary Key (SYSKEY)

INVOKE generates a host variable declaration for each column specified in the CREATE TABLE or CREATE VIEW statement that created the table or view definition. Therefore, if a system-defined primary key (SYSKEY) is specified for a view, INVOKE generates a host variable for SYSKEY. If SYSKEY is not specified for a view, INVOKE does not generate a SYSKEY host variable. (INVOKE does not generate a column for the SYSKEY of a table because SYSKEY cannot be specified in a CREATE TABLE statement.)

For example, suppose that you create a table by using this statement:

```
CREATE TABLE TYPESTAB (COLUMN-A INT, COLUMN-B INT)
```

INVOKE generates host variables for columns COLUMN-A and COLUMN-B but does not generate a host variable for the SYSKEY automatically associated with the table.

However, suppose that you create a view from the base table with a column corresponding to the table's SYSKEY:

```
CREATE VIEW AVIEW (COLUMN-X, COLUMN-Y, COLUMN-Z)
    AS SELECT SYSKEY, COLUMN-A, COLUMN-B
    FROM TYPESTAB
```

INVOKE generates host variables for columns COLUMN-X, COLUMN-Y, and COLUMN-Z because these columns are included in the view definition.

If the view definition does not specifically include the SYSKEY column, INVOKE does not generate a host variable for SYSKEY as shown in the next example:

```
CREATE VIEW AVIEW
    AS SELECT * FROM TYPESTAB
```

INVOKE generates host variables for only COLUMN-A and COLUMN-B because a SYSKEY column was not included in the view definition. The SELECT * statement acts in the same manner as an INVOKE directive, that is, it only selects columns explicitly specified in a CREATE statement.

Date-Time and INTERVAL Data Types

In an INVOKE directive, use the DATEFORMAT clause to specify the format of date-time (DATETIME, DATE, TIME, TIMESTAMP) or INTERVAL columns. SQL/MP converts the columns to character fields.

The size of each character field is determined by the size of the range of the fields defined for the data type. For example, if the range of fields for the column is YEAR TO DAY, the field in the invoked record description is 10 characters wide. Some examples follow.

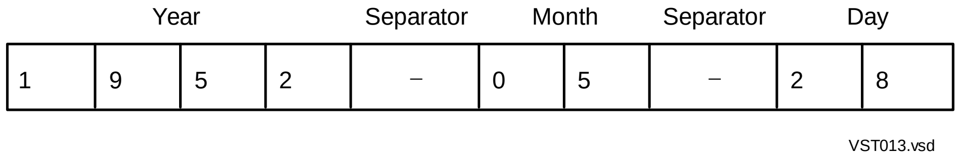
DATE Representation

Suppose that an SQL table has these column definitions:

```
NAME        CHAR(18)
BIRTH_DATE  DATE
```

Figure 2-1 on page 2-20 illustrates how a date is represented. The date is May 28, 1952.

Figure 2-1. DATE Representation



INVOKE generates this record description in the COBOL format:

```
01   EMPLOYEE
     02 NAME                PIC X(18)   VALUE SPACES.
     02 BIRTH-DATE          PIC X(10)   VALUE SPACES.
```

The host variable BIRTH-DATE is referenced in a program:

```
:BIRTH-DATE TYPE AS DATE
```

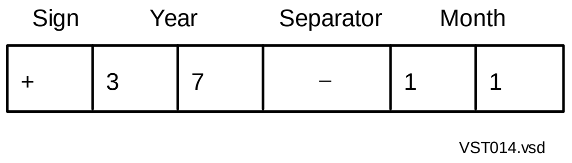
INTERVAL Representation

Suppose that an SQL table has these column definitions:

```
NAME    CHAR(18)
AGE     INTERVAL YEAR(2) TO MONTH
```

Figure 2-2 displays how an interval is represented. The age represented is 37 years, 11 months.

Figure 2-2. INTERVAL Representation



INVOKE generates this record description in the COBOL format:

```
01 EMPLOYEE
   02 NAME                  PIC X(18)   VALUE SPACES.
   02 AGE                   PIC X(6)    VALUE SPACES.
```

The host variable AGE is referenced in a program as follows:

```
:AGE TYPE AS INTERVAL YEAR(2) TO MONTH
```

Example—Creating DATETIME and INTERVAL Data Types

Example 2-3 on page 2-21 creates valid DATETIME data types. You can create INTERVAL data types similarly.

Example 2-3. Creating Valid DATETIME and INTERVAL Data Types

```
?INSPECT
?SYMBOLS
 IDENTIFICATION DIVISION.
 PROGRAM-ID.
      COBEXT.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. TANDEM/16.
 OBJECT-COMPUTER. TANDEM/16.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.

 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 OUTPUT-VAL PIC X(30).
 01 INPUT-VAL PIC X(30) VALUE SPACES.
 01 TEMP-STMT-TEXT PIC X(200) VALUE SPACES.
 EXEC SQL END DECLARE SECTION END-EXEC.

 EXEC SQL INCLUDE SQLCA END-EXEC.
?NOLIST
 EXTENDED-STORAGE SECTION.
?LIST

 PROCEDURE DIVISION.

 1000-DRIVER.
 PERFORM 3000-SPECIFY-ERROR-HANDLING.
 PERFORM 3100-PROCESS-QUERIES.
 STOP RUN.

 3000-SPECIFY-ERROR-HANDLING.
 EXEC SQL WHENEVER SQLERROR PERFORM :6000-HANDLE-ERROR END-EXEC.

 3100-PROCESS-QUERIES.

* Table DT has column TS of datatype DATETIME
 MOVE "2004-01-22:13:40:05.550000" TO INPUT-VAL
 MOVE "SELECT CAST(TS AS CHAR(29)) FROM dt where ts
-     " >=  CAST( CAST(? AS CHAR(29)) AS DATETIME
-     " YEAR TO FRACTION)" TO TEMP-STMT-TEXT

 EXEC SQL BEGIN WORK END-EXEC
 EXEC SQL PREPARE S1 FROM :TEMP-STMT-TEXT END-EXEC
 EXEC SQL DECLARE C1 CURSOR FOR S1 END-EXEC

 EXEC SQL OPEN C1 USING :INPUT-VAL END-EXEC

 PERFORM UNTIL SQLCODE < 0 OR SQLCODE = 100

 EXEC SQL FETCH C1 INTO :OUTPUT-VAL END-EXEC

 IF SQLCODE >= 0 AND SQLCODE NOT = 100 THEN
    DISPLAY "VALUE IS " OUTPUT-VAL
 END-IF

 END-PERFORM

 EXEC SQL CLOSE C1 END-EXEC
 EXEC SQL COMMIT WORK END-EXEC.

 6000-HANDLE-ERROR.
 ENTER TAL "SQLCADISPLAY" USING SQLCA
 STOP RUN.
```

Using Indicator Variables With the INVOKE Directive

The INVOKE directive automatically generates a two-byte indicator variable (data type short) for each host variable that corresponds to a column that allows a null value. The name of the indicator variable is the same name as the corresponding column plus a prefix, if you specify one, and a suffix. If you do not specify a prefix or suffix, INVOKE appends the default suffix -I to the name.

If a column name is 30 characters and the default indicator suffix -I is used, the -I is truncated, and the indicator variable name is then identical to the corresponding host variable name. To prevent this problem, use the PREFIX or NULL STRUCTURE clause for column names that are 30 or 31 characters.

The format of the indicator variable name depends on the PREFIX, SUFFIX, and NULL STRUCTURE clauses.

PREFIX and SUFFIX Clauses

The PREFIX and SUFFIX clauses causes INVOKE to generate an indicator variable name derived from the column name and the prefix or suffix. A default suffix of -I applies if the INVOKE directive omits these clauses. For example, if the column name is RETIRE-DATE, the format of the indicator variable is RETIRE-DATE-I.

You can specify a suffix other than an I by specifying a SUFFIX clause with the INVOKE statement. Or, you can replace the suffix with a prefix of your choosing by specifying the PREFIX clause.

This INVOKE directive specifies both a prefix and a suffix for the indicator variables:

```
EXEC SQL INVOKE BTABLE PREFIX I- SUFFIX -END  END-EXEC.
```

The HP COBOL compiler generates this structure:

```
*  Record Definition for table \SYS1.$VOL1.SUBV1.BTABLE
*  Definition current at 12:41:14 - 06/11/94
 01 BTABLE.
    02 I-ZCHAR-NULL-OK-END             PIC S9(4) COMP.
    02 ZCHAR-NULL-OK                   PIC X(10).
    02 I-ZNUM-NULL-OK-END              PIC S9(4) COMP.
    02 ZNUM-NULL-OK                    PIC S9(4) COMP.
```

NULL STRUCTURE Clause

The NULL STRUCTURE clause causes INVOKE to generate a group item for columns that contain the indicator variables. The group item name is derived from the column name. For example, a column named RETIRE-DATE has this format:

```
02 RETIRE-DATE.
   03 INDICATOR            PIC S9(4) COMP.
   03 VALUE                PIC X(10).
```

INDICATOR is the indicator variable. The SQL data type of an indicator variable is SMALLINT. The corresponding COBOL data type is PIC S9(4) COMP. VALUE is the host variable corresponding to the column value.

This example uses the NULL STRUCTURE clause, which causes columns that can contain null values to be declared as group items. This INVOKE directive contains a NULL STRUCTURE clause:

```
EXEC SQL INVOKE BTABLE NULL STRUCTURE  END-EXEC.
```

The generated record description is:

```
*  Record Definition for table \SYS1.$VOL1.SUBV1.BTABLE
*  Definition current at 12:41:11 - 06/11/94
 01 BTABLE.
    02 ZCHAR-NULL-OK.
       03 INDICATOR                        PIC S9(4) COMP.
       03 VALUE                            PIC X(10).
    02 ZNUM-NULL-OK.
       03 INDICATOR                        PIC S9(4) COMP.
       03 VALUE                            PIC S9(4) COMP.
```

Example 2-4 on page 2-24 declares and uses qualified host variable names and indicator variable names and shows the following:

- Host variable declaration with an INVOKE statement that specifies the suffix -I for indicator variables. The invoked record declaration is included as a comment in the example.
- Host variable indicator variable used in the SELECT statement. The columns that might contain a null value require the indicator variable following the host variable to receive information about null values.
- Indicator variable testing for a possible null value. If the value of the indicator variable following the select is less than 0, the associated column's value is NULL.

The example retrieves four columns of an order detail table. The table is similar to the ODETAIL table of the sample database except that the UNIT_PRICE and QTY_ORDERED columns allow null values.

Example 2-4. Using Host and Indicator Variable Names

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
   EXEC SQL
     INVOKE ODETAIL AS ORDER-DETAIL-RECORD SUFFIX -I
   END-EXEC.
*Record Description *****************************
*01 ORDER-DETAIL-RECORD.
*   02 ORDERNUM              PIC 9(6) COMP.
*   02 PARTNUM               PIC 9(4) COMP.
*   02 UNIT-PRICE-I          PIC S9(4) COMP.
*   02 UNIT-PRICE            PIC S9(6)V9(2) COMP.
*   02 QTY-ORDERED-I         PIC S9(4) COMP.
*   02 QTY-ORDERED           PIC 9(5) COMP.
 EXEC SQL END DECLARE SECTION  END-EXEC.
 ...
PROCEDURE DIVISION.
   ...
EXEC SQL
  SELECT ORDERNUM, PARTNUM, UNIT_PRICE, QTY_ORDERED
  INTO :ORDERNUM OF ORDER-DETAIL-RECORD,
       :PARTNUM OF ORDER-DETAIL-RECORD,
       :UNIT-PRICE OF ORDER-DETAIL-RECORD
          INDICATOR :UNIT-PRICE-I OF ORDER-DETAIL-RECORD,
           :QTY-ORDERED OF ORDER-DETAIL-RECORD
            INDICATOR :QTY-ORDERED-I OF ORDER-DETAIL-RECORD,
          FROM SALES.ODETAIL
          WHERE ORDERNUM = 300380 AND PARTNUM = 2402
END-EXEC.
...
IF UNIT-PRICE-I OF ORDER-DETAIL-RECORD   < 0
      OR QTY-ORDERED-I OF ORDER-DETAIL-RECORD  < 0 THEN
      PERFORM 0500-HANDLE-NULL-VALUE
ELSE  PERFORM 0300-DISPLAY-RESULT.
...
```

Using INVOKE With SQLCI

You can also execute the INVOKE directive interactively through SQLCI to create host variable declarations in a copy file. For example, this INVOKE directive generates a COBOL copy file from the DEPT table:

```
>> INVOKE =DEPT FORMAT COBOL85 TO COPYLIB (DEPTREC);
   ...
```

Use the COBOL SOURCE directive to copy the host variable declarations in your program's compilation unit:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
?SOURCE COPYLIB (DEPTREC)
EXEC SQL END DECLARE SECTION END-EXEC.
```

Using INVOKE with SQLCI provides less program independence than embedding INVOKE in your program, because you must re-create the host variable declarations if the referenced table changes. However, if necessary, you can edit the host variables before copying them into your program's compilation unit.

Associating a Character Set With a Host Variable

By default, SQL/MP associates a single-byte unknown character set with a host variable. To associate a specific character set such as ISO 8859/*n*, Kanji, or KSC5601 with a host variable, include the CHARACTER SET clause in the host variable declaration using this syntax:

```
level  host-variable
  [ CHARACTER SET [ IS ] "character-set-name" ]
    PIC[TURE] { X [ ( length ) ] }...[ COBOL-clause ]... .
```

level

is the COBOL level number.

host-variable

is a COBOL identifier that is the name of the host variable, which must conform to COBOL naming conventions.

"*character-set-name*"

specifies the name of the character set, which must be one of these keywords:

ISO8859*n* (where n ranges from 1 through 9)
KANJI
KSC5601
UNKNOWN

You must enclose *character-set-name* in double quotation marks ("). Any leading or trailing spaces inside the double quotation marks are ignored.

The UNKNOWN keyword indicates an unknown single-byte character set and is equivalent to omitting the CHARACTER SET clause.

length

is the length in characters (not bytes) of the host variable. For a double-byte character set, you must code the PICTURE clause specification (the X part) on a single line.

COBOL-clause

is a COBOL clause such as VALUE or USAGE. For a description of the COBOL clauses, see the *COBOL85 for NonStop Systems Manual*.

Treatment in COBOL Statements

A COBOL statement treats a host variable declared with the CHARACTER SET clause as if the host variable had been declared without the clause. The total length of the host variable is the length in the PICTURE clause multiplied by the number of bytes per character for the specified character set.

For example, the total length of the first two declarations in this table is the same as the length in the PICTURE clause. However, the total length of the third declaration is twice the length in the PICTURE clause because KANJI is a double-byte character set.

Host Variable Declaration	**Treatment in COBOL Statement**
`77 HVAR-1 CHARACTER SET "ISO88591" PIC X(5).`	`77 HVAR-1 PIC X(5).`
`77 HVAR-1 CHARACTER SET "ISO88591" PIC X(10).`	`77 HVAR-1 PIC X(10).`
`77 HVAR-2 CHARACTER SET "KANJI" PIC X(10).`	`77 HVAR-2 PIC X(20).`

VARCHAR Data Type

If you specify the CHARACTER SET clause with a host variable declared as a VARCHAR data type, you must set the length data item (LEN) of the VARCHAR group item to the host variable length in bytes and not characters. For example, this host variable declaration uses the double-byte KSC5601 character set. The MOVE statement sets the length (LEN OF EMPLOYEE-NAME) of the host variable name to 16, because the name (VAL OF EMPLOYEE-NAME) contains 8 double-byte characters (represented as “c1c2c3c4c5c6c7c8”).

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMPLOYEE-NAME.
  02 LEN PIC S9(4) COMP.
  02 VAL CHARACTER SET "KSC5601" PIC X(10).
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE-DIVISION.
...
MOVE "c1c2c3c4c5c6c7c8" TO VAL OF EMPLOYEE-NAME.
MOVE 16 TO LEN OF EMPLOYEE-NAME.
EXEC SQL
  INSERT INTO EMPLOYEE VALUES (:EMPLOYEE-NAME)
END-EXEC.
...
```

3 SQL/MP Statements and Directives

This section describes NonStop SQL/MP statements and directives you can embed in a COBOL program. For a detailed description, including the syntax, of all SQL statements and directives, see the *SQL/MP Reference Manual*.

Topics include:

- Embedding SQL Statements
- Finding Information on page 3-3

Embedding SQL Statements

Use this syntax to embed an SQL statement or directive in a COBOL source file:

```
EXEC SQL  sql-statement-or-directive  END-EXEC.
```

sql-statement-or-directive

is any SQL statement or directive shown in Table 3-1, NonStop SQL/MP Statements and Directives, on page 3-3. The statement or directive must begin with the keywords EXEC SQL and end with END-EXEC. The EXEC SQL keywords do not require a hyphen or period, but the END-EXEC keywords require both the hyphen and period. (However, a period after the END-EXEC keywords after the BEGIN DECLARE SECTION or END DECLARE SECTION directive is ignored.)

Coding SQL Statements and Directives

In general, handle embedded SQL statements and directives as if they were COBOL statements. Follow the same formatting and line continuation conventions for SQL statements as you use for COBOL statements. Here are a some specific guidelines to follow when you embed SQL statements and directives in a COBOL program:

- Code an SQL statement or directive on a single source code line or over several lines:

```
EXEC SQL WHENEVER SQLERROR PERFORM :HANDLE-ERROR END-EXEC.

EXEC SQL DROP TABLE \NY.$DISK1.INVENT.SUPPLIER END-EXEC.

EXEC SQL
  SELECT CUSTOMER.CUSTNAME
    INTO :CUSTOMER.CUSTNAME
      FROM =CUSTOMER
        WHERE CUSTNUM = :FIND_THIS_CUSTOMER
END-EXEC.
```

- Do not nest SQL statements or directives.

- Precede an SQL statement with a COBOL section or paragraph name.
- If you place COBOL and SQL statements on the same line, these restrictions apply:
    - The COBOL statement cannot be a COPY or REPLACE statement.
    - A COBOL statement must follow the embedded SQL statement terminator. It cannot precede an embedded SQL statement on a line.
- Use either SQL or COBOL comments within SQL statements and directives. An SQL comment begins with a double hyphen (--) and ends with the end of the line. A COBOL comment has an asterisk (*) in the indicator field (column 1 in HP format or column 7 in ANSI format) and ends with the end of the line.
- If you specify a delimiter on an executable SQL statement in the Procedure Division, the delimiter affects program execution. A COBOL statement is generated from the embedded SQL statement, and the delimiter is appended to the generated statement. Consequently, you can use SQL statements in conditional statements, such as IF and ELSE.
- If you specify a delimiter on an SQL statement in the Data Division or a non-executable statement in the Procedure Division, the delimiter appears only in a comment line. The HP COBOL compiler copies both the embedded SQL directive and the delimiter to comment lines. The delimiter is not appended to any resulting data declaration.
- Do not code an SQL statement or directive on a COBOL debugging line.

Placing SQL Statements and Directives

Place SQL statements, SQL directives, and COBOL compiler directives in a COBOL source file as described in this subsection.

SQL Compiler Directive

You must specify the SQL compiler directive before the first Identification Division in your program. The SQL directive indicates to the HP COBOL compiler that the compilation unit contains SQL statements or directives.

You can specify the SQL directive either in your source code file or as a compiler option in the implicit TACL RUN command for the HP COBOL compiler. This example shows the SQL directive in a source code file:

```
?SQL
```

This example shows the SQL directive as a compiler option:

```
COBOL85 /IN COBSRC,OUT $S.#COBLIST,NOWAIT/ COBOBJ; SQL
```

After the SQL directive, place other SQL statements and directives in a COBOL source file as described in this subsection.

Data Division

You can use these statements and directives in the Data Division:

- BEGIN DECLARE SECTION and END DECLARE SECTION directives
- DECLARE CURSOR statement
- INVOKE directive
- INCLUDE STRUCTURES directive
- INCLUDE SQLCA, INCLUDE SQLDA, and INCLUDE SQLSA directives
- INCLUDE SQLCODEX directive

Procedure Division

You can use these statements and directives in the Procedure Division:

- Data Control Language (DCL) statements
- Data Definition Language (DDL) statements
- Data Manipulation Language (DML) statements (including DECLARE CURSOR)
- Data Status Language (DSL) statements
- Dynamic SQL statements
- WHENEVER directive

Anywhere in the Program

You can use the CONTROL directive anywhere in a program.

Finding Information

Table 3-1 summarizes the SQL/MP statements and directives you can embed in a COBOL program and shows where each statement or directive is documented.

Table 3-1. NonStop SQL/MP Statements and Directives (page 1 of 5)

Statement or Directive	Manual*	Description
Data Declaration Directives		
BEGIN DECLARE SECTION	SQLRM, COBPM	Designates the beginning of host variable declarations.
END DECLARE SECTION	SQLRM, COBPM	Designates the end of host variable declarations.
INCLUDE STRUCTURES	SQLRM, COBPM	Specifies the version of SQL structures generated.
INCLUDE SQLCA	SQLRM, COBPM	Generates the SQLCA structure for run-time status and error information.

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

Table 3-1. NonStop SQL/MP Statements and Directives (page 2 of 5)

Statement or Directive	Manual*	Description
Data Declaration Directives		
INCLUDE SQLCODEX	SQLRM, COBPM	Enables declaring level-88 items to check for specified conditions.
INCLUDE SQLDA	SQLRM, COBPM	Generates the SQLDA structure to receive information about input and output variables for dynamic SQL statements.
INCLUDE SQLSA	SQLRM, COBPM	Generates the SQLSA structure to receive execution statistics about DML or PREPARE statements.
INVOKE	SQLRM, COBPM	Generates a COBOL record description of a table or view.
Data Definition Language (DDL) Statements		
ALTER CATALOG	SQLRM	Alters the security attributes of a catalog.
ALTER COLLATION	SQLRM	Alters the security attributes of a collation; renames a collation.
ALTER INDEX	SQLRM	Alters security attributes of indexes; alters physical file attributes of indexes and partitions of indexes; adds and drops partitions; renames indexes and partitions.
ALTER PROGRAM	SQLRM	Alters security attributes for a program; renames a program.
ALTER TABLE	SQLRM	Alters security attributes of tables; alters physical file attributes of tables and partitions of tables; alters the HEADING attribute for columns of tables and views; adds and drops table partitions; renames tables and partitions of tables; adds new columns to tables.
ALTER VIEW	SQLRM	Alters security attributes for a view or renames a view.
COMMENT	SQLRM	Adds a comment to an object definition.
CREATE	SQLRM	Creates a collation, constraint, catalog, index, table, or view.
DROP	SQLRM	Drops a collation, constraint, catalog, index, program, table, or view.

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

Table 3-1. NonStop SQL/MP Statements and Directives (page 3 of 5)

Statement or Directive	Manual*	Description
Data Definition Language (DDL) Statements		
HELP TEXT	SQLRM	Specifies help text for a column of a table or view.
UPDATE STATISTICS	SQLRM	Updates information about the contents of a table and its indexes.
Data Manipulation Language (DML) Statements		
CLOSE	SQLRM, COBPM	Terminates a cursor.
DECLARE CURSOR	SQLRM, COBPM	Defines a cursor.
DELETE	SQLRM, COBPM	Deletes rows from a table or view.
FETCH	SQLRM, COBPM	Retrieves a row from a cursor.
INSERT	SQLRM, COBPM	Inserts rows into a table or view.
OPEN	SQLRM, COBPM	Opens a cursor.
SELECT	SQLRM, COBPM	Retrieves data from tables and views.
UPDATE	SQLRM, COBPM	Updates values in columns of a table or view.
Data Control Language (DCL) Statements		
CONTROL EXECUTOR	SQLRM, COBPM	Specifies whether to process data using a single executor or multiple executors working in parallel.
CONTROL QUERY	SQLRM, COBPM	Specifies whether to optimize query time for the first few rows or for all rows, whether to consider a hash join algorithm for executing queries, or whether to use execution-time name resolution.
CONTROL TABLE	SQLRM, COBPM	Specifies parameters that control locks, opens, buffers, access paths, join methods, and join sequences on tables and views.

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

Table 3-1. NonStop SQL/MP Statements and Directives (page 4 of 5)

Statement or Directive	Manual*	Description
Data Control Language (DCL) Statements		
FREE RESOURCES	SQLRM	Closes cursors and releases locks held by the program.
LOCK TABLE	SQLRM	Locks a table or underlying tables of a view and associated indexes.
UNLOCK TABLE	SQLRM	Releases locks held on nonaudited tables and views.
Data Status Language (DSL) Statements		
GET CATALOG OF SYSTEM	SQLRM	Returns the name of a local or remote system catalog.
GET VERSION	SQLRM, COBPM	Returns the version of a catalog, collation, index, table, or view; also returns the version of the SQL/MP system software.
GET VERSION OF PROGRAM	SQLRM, COBPM	Returns the program catalog version (PCV), program format version (PFV), or host object SQL version (HOSV) of an SQL program file.
Dynamic SQL Statements		
DESCRIBE	SQLRM, COBPM	Returns information about output variables in prepared statements.
DESCRIBE INPUT	SQLRM, COBPM	Returns information about input variables in prepared statements.
EXECUTE	SQLRM, COBPM	Executes a prepared statement.
EXECUTE IMMEDIATE	SQLRM, COBPM	Executes an SQL statement contained in a host variable.
PREPARE	SQLRM, COBPM	Compiles a DDL, DML, DCL, or DSL statement.
RELEASE	SQLRM	Deallocates memory for a dynamic SQL statement referred to through a host variable.
Error-Checking Directive		
WHENEVER	SQLRM, COBPM	Generates code that checks SQL statement execution for errors, warnings, and the not found condition for rows.

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

Table 3-1. NonStop SQL/MP Statements and Directives (page 5 of 5)

Statement or Directive	Manual*	Description
Transaction Control Statements		
BEGIN WORK	SQLRM	Starts a TMF transaction.
COMMIT WORK	SQLRM	Commits all database changes made during the current TMF transaction and frees resources.
ROLLBACK WORK	SQLRM	Backs out the current TMF transaction and frees resources.

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

Table 3-2 summarizes COBOL compiler directives that apply to a COBOL program containing embedded SQL statements and directives. For a description of all other COBOL compiler directives, see the *COBOL85 for NonStop Systems Manual*.

Table 3-2. COBOL Compiler Directives for SQL/MP

Directive	Manual *	Description
SOURCE	COBPM, COBRM	Copies source code from a source file into the compilation.
SQL	COBPM, COBRM	Indicates to the HP COBOL compiler that a program contains embedded SQL statements or directives. Specifies options for processing the SQL statements or directives: ● PAGES specifies the number of 2048-byte pages that the HP COBOL compiler allocates to the SQL compiler interface (SCI). ● SQLMAP generates an SQL map in the listing. ● WHENEVERLIST writes active WHENEVER options to the listing file after each SQL statement is processed. ● RELEASE1 or RELEASE2 specifies the version of the SQL/MP features in the program (including the SQL data structures) and the version of SQL/MP software on which the program can run.
SQLMEM	COBPM, COBRM	Controls the placement of SQL internal structures in either the user data segment (Working-Storage Section) or extended data segment (Extended-Storage Section).

* This statement is documented in one or more of these manuals:

SQLRM	*SQL/MP Reference Manual*
COBPM	*SQL/MP Programming Manual for COBOL*

4 Data Retrieval and Modification

This section describes how to access data in a NonStop SQL/MP database by using the Data Manipulation Language (DML) statements in a COBOL program.

Topics include:

Table 4-1 provides some guidelines for using these statements.

Table 4-1. SQL/MP Statements for Data Retrieval and Modification

NonStop SQL/MP Statement	Description
Single-Row SELECT statement	Retrieves a single row of data from a table or protection view and places the specified column values in host variables. Use when you need to retrieve only a single row.
SELECT statement with a cursor	Retrieves a set of rows from a table or view, one row at a time, and places the specified column values in host variables. Use when you need to retrieve more than one row.
INSERT statement	Inserts one or more rows into a table or protection view. Use for all INSERT operations.
UPDATE statement without a cursor	Updates the values in one or more columns in a single row or a set of rows of a table or protection view. Use when you do not need to test a column value in a row before you update the row.
UPDATE statement with a cursor	Updates the values in one or more columns in a set of rows, one row at a time. Use when you need to test a column value in a row before you update the row.
DELETE statement without a cursor	Deletes a single row or a set of rows from a table or protection view. Use when you do not need to test a column value in a row before you delete the row.
DELETE statement with a cursor	Deletes a set of rows, one row at a time, from a table or protection view. Use when you need to test a column value in a row before you delete the row.

Note. Using a cursor can sometimes degrade a program's performance. A cursor operation requires three statements (OPEN, FETCH, and CLOSE), which increase the messages between the file system and disk process. Therefore, consider not using a cursor if a single-row SELECT statement is sufficient.

Opening and Closing Tables and Views

SQL/MP automatically opens and closes tables and views during the execution of DDL statements, DML statements, and SQL utility operations such as a LOAD or COPY. SQL/MP opens a table or view when a host-language program executes the first SQL statement that refers to the table or view and then closes the table or view when the program that opened it stops. A program cannot explicitly open an SQL table or view. However, a program can force SQL/MP to close a table using the CLOSE TABLES option of the FREE RESOURCES statement.

By default, SQL/MP opens partitions of base tables and indexes only as they are needed by a program. To cause SQL/MP to open all indexes and partitions the first time any partition is accessed, use the OPEN ALL option of the CONTROL TABLE directive.

Note. Using the CONTROL TABLE statement with the OPEN ALL option could increase the amount of work done by an SQL statement. For efficient performance, use the OPEN ALL option with the CONTROL TABLE statement only if all these are true:

- When all open activity must occur when the program first starts (add a "dummy" call to the cursor during initialization).
- When the object containing the cursor will eventually access all partitions.
- When the plan for the cursor is not a parallel plan.

Causes of SQL Error 8204 (Lost Open Error)

SQL error 8204 is sometimes referred to as the “lost open” error. This scenario explains how this error can occur:

1. A program accesses a table or view by using one or more static DML statements (SELECT, INSERT, UPDATE, or DELETE) or a static cursor. The SQL executor opens the table or view for the program.
2. Any locks associated with the statements in Step 1 are released (for example, because the transaction ended). Another user then executes one of these DDL statements or utility operations for the table or view, which causes the system to terminate the program's open:
   - ALTER TABLE with ADD COLUMN, ADD PARTITION, DROP PARTITION, or RENAME
   - ALTER TABLE with AUDIT, LOCKLENGTH
   - ALTER INDEX with ADD PARTITION, DROP PARTITION, or RENAME

- ALTER INDEX with LOCKLENGTH
- ALTER VIEW with RENAME
- CREATE CONSTRAINT and CREATE INDEX
- DROP CONSTRAINT, DROP INDEX, DROP TABLE, or DROP VIEW (protection view only)
- UPDATE STATISTICS
- COPY, LOAD, PURGEDATA, or RESTORE utility operation

(A disk or network line that goes down and then comes back up can also cause the system to terminate a program's open.)

3. The program tries to execute another SQL statement for the table or view.
4. The SQL executor tries to recover, as described in the next subsection. However, if it cannot recover from the error, the executor returns error -8204 to the program, and the program loses its open for the table or view.

Recovering From SQL Error 8204

If a program executes a static DML statement and the open for a table or view it is using has been lost because of a DDL statement or utility operation, the SQL executor tries to recover as described in this subsection.

Simple DML Statements

For static DML statements (SELECT, INSERT, UPDATE, and DELETE), the SQL executor reopens the changed table or view and then retries the DML statement once using the new definition of the table or view. If the retry is successful, the SQL executor returns a warning (8204) to the program. However, if the retry fails, the SQL executor returns an error (-8204).

To recover from SQL error -8204 for a simple DML statement, a program might need to abort the transaction and restart the operation from its beginning.

Because some DDL changes can invalidate a DML statement, the SQL executor might first need to recompile the DML statement to use the new definition of the changed table or view. In some cases, the similarity check can prevent recompilation. For more information, see Section 8, Program Invalidation and Automatic SQL Recompilation.

If the program does not allow automatic recompilation (the NORECOMPILE option is set), the SQL executor returns error -8027. In this case, you must explicitly recompile the program by using the new definition of the table or view.

Static Cursor Operations

For a static cursor operation, the SQL executor tries to reestablish the open in these situations:

- The program has not yet opened the cursor.
- The program has opened the cursor, but the OPEN CURSOR statement did not require any input host variables, and the first FETCH statement has not yet been executed.

However, if the problem occurs on a FETCH statement, the SQL executor closes the cursor and returns error -8204. The program must then close and reopen the cursor before executing a subsequent FETCH statement. The program might need to abort the transaction and restart the cursor operation from its beginning.

Single-Row SELECT Statement

A single-row SELECT statement retrieves a single row of data from one or more tables or views and places the column values into corresponding host variables.

To select a set of rows one row at a time by using a cursor, see Using SQL Cursors on page 4-12.

To execute a SELECT statement, a process started by the program must have read access to all tables, protection views, and the underlying tables of shorthand views used in the statement. For information about process access, see Required Access Authority on page 7-1.

Do not use an asterisk (*) in a SELECT statement in a COBOL program. A SELECT statement with an asterisk always assigns columns in the result table from the current definition of the referenced tables or views. If columns have been added to a table, the retrieved data values might not be in the expected order.

SQL/MP returns these values to SQLCODE after a SELECT statement.

SQLCODE Value	Description
`0`	The SELECT statement was successful.
`100`	No rows qualified for the SELECT statement specification.
`< 0`	An error occurred; SQLCODE contains the error number.
`> 0 (not 100)`	A warning occurred; SQLCODE contains the warning number.

For more information about SQLCODE, see Section 9, Error and Status Reporting.

Using a Column Value to Select Data

This SELECT statement returns a row containing a customer’s name and address based on a unique column value (a nonkey value). Each customer is identified by a unique number so that only one customer satisfies the query. This example uses a WHERE clause to specify that the CUSTOMER.CUSTNAME column contains a unique value equal to the host variable named FIND-THIS-CUSTOMER. Example 4-1 sets FIND-THIS-CUSTOMER to customer number 5635 by using an assignment statement, but in a typical application, a user would enter the customer number.

SQL/MP scans the database to find the first row indicated by CUSTNUM and then returns this row to the program. Because CUSTNUM is not a primary key, SQL/MP also reads the remainder of the table to verify that the row returned is the only qualifying row. If it is not, SQL/MP returns an error.

Example 4-1. Using a Column Value to Select Data

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 CUSTOMER.
     02 CUSTNUM           PIC 9(4) DISPLAY.
     02 CUSTNAME          PIC X(18).
     02 STREET            PIC X(22).
     02 CITY              PIC X(14).
     02 STATE             PIC X(12).
     02 POSTCODE          PIC X(10).
  01 FIND-THIS-CUSTOMER   PIC 9(4)   VALUE  0.
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
100-BEGIN.
EXEC SQL WHENEVER NOT FOUND PERFORM :400-NOT-FOUND END-EXEC.

    MOVE 5635 TO FIND-THIS-CUSTOMER.
    EXEC SQL
      SELECT  CUSTNAME,  STREET,  CITY,  STATE,  POSTCODE
        INTO :CUSTNAME, :STREET, :CITY, :STATE, :POSTCODE
        FROM SALES.CUSTOMER
        WHERE CUSTNUM = :FIND-THIS-CUSTOMER
        BROWSE ACCESS
    END-EXEC.

    DISPLAY CUSTNAME, STREET, CITY, STATE, POSTCODE.

400-NOT-FOUND.
    PERFORM 5000-CLOSE-CURSOR.
    DISPLAY "CUSTOMER NOT FOUND: "FIND-THIS-CUSTOMER"
    PERFORM 8880-ABORT-TRANSACTION.
```

Using a Primary Key Value to Select Data

This SELECT statement returns an employee’s first name, last name, and department number from the EMPLOYEE table by using a primary key value (EMPNUM column). The WHERE clause specifies that the selected row contains a primary key with a value equal to the host variable named FIND-THIS-EMPLOYEE. The SELECT statement retrieves only one row because the primary key value is unique.

```
MOVE INPUT-EMPNAME TO FIND-THIS-EMPLOYEE.

EXEC SQL SELECT EMPLOYEE.FIRST-NAME,
                EMPLOYEE.LAST-NAME,
                EMPLOYEE.DEPTNUM
           INTO :EMPLOYEE.FIRST-NAME,
                :EMPLOYEE.LAST-NAME,
                :EMPLOYEE.DEPTNUM
           FROM PERSNL.EMPLOYEE
           WHERE EMPLOYEE.EMPNUM = :FIND-THIS-EMPLOYEE
END-EXEC.
```

Using IN SHARE MODE or IN EXCLUSIVE MODE

If you include the keyword IN in the EXCLUSIVE MODE or SHARE MODE option after a host variable, the statement generates a syntax error. For example, this statement generates a syntax error because HOST-VAR IN EXCLUSIVE MODE is interpreted as a host variable:

```
EXEC SQL SELECT COLUMN INTO :HOST-VAR FROM ATABLE
      WHERE COLUMN > :HOST-VAR IN EXCLUSIVE MODE END-EXEC.
```

To prevent the syntax error, omit the keyword IN:

```
EXEC SQL SELECT COLUMN INTO :HOST-VAR FROM ATABLE
      WHERE COLUMN > :HOST-VAR EXCLUSIVE MODE END-EXEC.
```

INSERT Statement

The INSERT statement inserts one or more rows into a table or protection view. To insert data, a program moves the new data to a series of host variables and then executes an INSERT statement to transfer the host variable values to the table.

To execute an INSERT statement, a process started by the program must have read and write access to the table or view receiving the data and read access to tables or views you include in a SELECT statement. For information about process access, see Required Access Authority on page 7-1.

SQL/MP returns these values to SQLCODE after an INSERT statement.

SQLCODE Value	Description
0	The INSERT statement was successful.
100	No rows qualified for an INSERT using a SELECT statement specification.
< 0	An error occurred; SQLCODE contains the error number.
> 0 (not 100)	A warning occurred; SQLCODE contains the first warning number.

If an INSERT statement executes successfully, the SQLCA structure contains the number of rows inserted. (If the INSERT statement fails, do not rely on the SQLCA structure for an accurate count of the number of rows inserted.) To return the contents of the SQLCA structure, use the SQLCA_DISPLAY2_ or SQLCA_TOBUFFER2_ procedure.

For more information, see Section 5, SQL/MP System Procedures, and Section 9, Error and Status Reporting.

Inserting a Single Row

This INSERT statement inserts a row (JOBCODE and JOBDESC columns) into the JOB table:

```
 EXEC SQL BEGIN DECLARE SECTION;
* Declare host variables HV-JOBCODE and HV-JOBDESC.
 ...
 EXEC SQL END DECLARE SECTION;
 ...
 PROCEDURE DIVISION.
 ...
* Move values to HV-JOBCODE and HV-JOBDESC.
 ...
 EXEC SQL INSERT INTO PERSNL.JOB (JOBCODE, JOBDESC)
                 VALUES (:HV-JOBCODE, :HV-JOBDESC)
 END-EXEC.
 ...
```

If the INSERT operation fails, check for SQL error 8227, which indicates you attempted to insert a row with an existing key (primary or unique alternate).

Inserting a Null Value

This example inserts a row into the EMPLOYEE table and sets the SALARY column to a null value using an indicator variable:

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Declare host variables EMPNUM, FIRST-NAME,
* LAST-NAME, DEPTNUM, JOBCODE, and SALARY.

 01 IND-1  PIC S9(4) COMP.
 ...
 EXEC SQL END DECLARE SECTION END-EXEC.
 ...

 PROCEDURE DIVISION.
 ...
 MOVE -1 TO IND-1.
* Move values to host variables EMPNUM, FIRST-NAME,
* LAST-NAME, DEPTNUM, JOBCODE, and SALARY.
 ...
 EXEC SQL INSERT INTO PERSNL.EMPLOYEE
          VALUES (:EMPNUM, :FIRST-NAME, :LAST-NAME,
                  :DEPTNUM,:JOBCODE,
                  :SALARY INDICATOR :IND-1)
 END-EXEC.
```

This example uses the NULL keyword instead of an indicator variable:

```
 EXEC SQL INSERT INTO PERSNL.EMPLOYEE
          VALUES (:EMPNUM, :FIRST-NAME, :LAST-NAME,
                  :DEPTNUM,:JOBCODE, NULL)
 END-EXEC.
```

Inserting a Timestamp

This example inserts a timestamp value into COLUMNA of TABLET. The COLUMNA definition specifies the data type TIMESTAMP DEFAULT CURRENT. This example uses the JULIANTIMESTAMP and CONVERTTIMESTAMP system procedures and the SQL CONVERTTIMESTAMP function.

```
 EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
   01 DATETIME     PIC S9(18) COMP.
 EXEC SQL END DECLARE SECTION  END-EXEC.

 PROCEDURE DIVISION.
 MAIN-DRIVER.
 ...
* Get current Julian timestamp in Greenwich mean time.
 ENTER TAL "JULIANTIMESTAMP" GIVING DATETIME.
* Convert timestamp to local time.
 ENTER TAL "CONVERTTIMESTAMP" USING DATETIME GIVING DATETIME.
 ...
* Insert value into COLUMNA of TABLET.
 EXEC SQL INSERT INTO TABLET (COLUMNA)
                 VALUES ( CONVERTTIMESTAMP (:DATETIME))
 END-EXEC.
```

UPDATE Statement

The UPDATE statement updates the values in one or more columns in a single row or a set of rows of a table or protection view. (To update a set of rows one row at a time by using a cursor, see Using SQL Cursors on page 4-12.)

To execute an UPDATE statement, a process started by the program must have read and write access to the table or view being updated and read access to any table or view specified in subqueries of the search condition. For information about process access, see Required Access Authority on page 7-1.

For audited tables and views, SQL/MP holds a lock on an updated row until the TMF transaction is committed or rolled back. For a nonaudited table, SQL/MP holds the lock until the program releases it.

SQL/MP returns these values to SQLCODE after an UPDATE statement.

SQLCODE Value	Description
0	The UPDATE statement was successful.
100	No rows were found on a search condition.
< 0	An error occurred; SQLCODE contains the error number.
> 0 (not 100)	A warning occurred; SQLCODE contains the first warning number.

The UPDATE statement updates rows in sequence. If an error occurs, SQL/MP returns an error code to SQLCODE and terminates the UPDATE operation. The SQLCA structure contains the number of rows updated. (If the UPDATE statement fails, do not rely on the SQLCA structure for an accurate count of the number of rows updated.) To return the contents of the SQLCA structure, use the SQLCA_DISPLAY2_ or SQLCA_TOBUFFER2_ procedure.

For more information, see Section 5, SQL/MP System Procedures, and Section 9, Error and Status Reporting.

Updating a Single Row

This example updates a single row of the ORDERS table that contains information about the order number specified by UPDATE-ORDERNUM. In a typical application, a user enters the values for UPDATE-DATE and UPDATE-ORDERNUM. If the UPDATE operation fails, check for SQL error 8227, which indicates you attempted to update a row with an existing key (primary or unique alternate).

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
  01 ORDERS.
     02 ORDERNUM     PIC 9(6) DISPLAY.
     02 ORDER-DATE   PIC 9(6) DISPLAY
     02 DELIV-DATE   PIC 9(6) DISPLAY.
     02 SALESREP     PIC 9(4) DISPLAY.
     02 CUSTNUM      PIC 9(4) DISPLAY.
  01 NEWDATE         PIC 9(6) DISPLAY.
EXEC SQL END DECLARE SECTION  END-EXEC.
...
PROCEDURE DIVISION.
...
MOVE UPDATE-DATE TO NEWDATE.
MOVE UPDATE-ORDERNUM TO ORDERNUM OF ORDERS.
EXEC SQL UPDATE ORDERS SET DELIV-DATE = :NEWDATE
    WHERE ORDERNUM = :ORDERNUM OF ORDERS
    STABLE ACCESS END-EXEC.
```

Updating Multiple Rows

If you do not need to check a value in a row before you update the row, use a single UPDATE statement to update multiple rows in a table. This example updates the SALARY column of all rows in the EMPLOYEE table where the SALARY value is less than HOSTVAR-MIN-SALARY. A user enters the values for HOSTVAR-INC and HOSTVAR-MIN-SALARY.

```
EXEC SQL UPDATE PERSNL.EMPLOYEE
         SET SALARY = SALARY * :HOSTVAR-INC
         WHERE SALARY < :HOSTVAR-MIN-SALARY END-EXEC.
```

This example updates all rows in the EMPLOYEE. DEPTNUM column that contain the value in HOSTVAR-OLD-DEPTNUM. After the update, all employees who were in the department specified by HOSTVAR-OLD-DEPTNUM are moved to the department specified by HOSTVAR-NEW-DEPTNUM. A user enters the values for HOSTVAR-OLD-DEPTNUM and HOSTVAR-NEW-DEPTNUM.

```
EXEC SQL UPDATE PERSNL.EMPLOYEE
         SET DEPTNUM = :HOSTVAR-NEW-DEPTNUM
         WHERE DEPTNUM = :HOSTVAR-OLD-DEPTNUM END-EXEC.
```

Updating Columns With Null Values

This example updates the specified SALARY column in the EMPLOYEE table to a null value using an indicator variable. The SET-TO-NULLS host variable specifies the row to update.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
EXEC SQL INVOKE PERSNL.EMPLOYEE AS EMP-TBL END-EXEC.
...
01 IND-1     PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
MOVE -1 TO IND-1.
MOVE NULL-JOBCODE TO SET-TO-NULLS.
EXEC SQL UPDATE PERSNL.EMPLOYEE
         SET SALARY =:SALARY OF EMP-TBL INDICATOR :IND-1
         WHERE JOBCODE OF EMP-TBL = :SET-TO-NULLS END-EXEC.
```

This example uses the NULL keyword instead of an indicator variable:

```
EXEC SQL UPDATE PERSNL.EMPLOYEE
         SET SALARY = NULL
         WHERE JOBCODE = :SET-TO-NULLS END-EXEC.
```

DELETE Statement

The DELETE statement deletes one or more rows from a table or protection view. If you delete all rows from a table, the table still exists until it is deleted from the catalog by a DROP TABLE statement. (To delete a set of rows one row at a time by using a cursor, see Using SQL Cursors on page 4-12.)

To execute a DELETE statement, a process started by the program must have read and write access to the table or view that contains the rows to be deleted and to tables or views in subqueries of the search condition. For information about process access, see Required Access Authority on page 7-1.

SQL/MP returns these values to SQLCODE after a DELETE statement.

SQLCODE Value	Description
`0`	The DELETE statement was successful.
`100`	No rows were found on a search condition.
`< 0`	An error occurred; SQLCODE contains the error number.
`> 0 (not 100)`	A warning occurred; SQLCODE contains the first warning number.

After a successful DELETE operation, the SQLCA structure contains the number of rows deleted. If an error occurs on a DELETE operation, the SQLCA contains the approximate number of rows deleted. To return the contents of the SQLCA, use SQLCA_DISPLAY2_ or SQLCA_TOBUFFER2_ procedure.

For more information, see Section 5, SQL/MP System Procedures, and Section 9, Error and Status Reporting.

Deleting a Single Row

To delete a single row, move a key value to a host variable and then specify the host variable in the WHERE clause. This DELETE statement deletes only one row of the EMPLOYEE table because each value in EMPNUM (the primary key) is unique. A user enters the value for the host variable named HOSTVAR-EMPNUM.

```
EXEC SQL DELETE FROM PERSNL.EMPLOYEE
     WHERE EMPNUM = :HOSTVAR-EMPNUM END-EXEC.
```

Deleting Multiple Rows

If you do not need to check a column value before you delete a row, use a single DELETE statement to delete multiple rows in a table. This example deletes all rows (or employees) from the EMPLOYEE table specified by DELETE-DEPTNUM (which is entered by a user).

```
EXEC SQL DELETE FROM PERSNL.EMPLOYEE
     WHERE DEPTNUM = :DELETE-DEPTNUM END-EXEC.
```

This example deletes all suppliers from the PARTSUPP table who charge more than TERMINAL-MAX-COST for a terminal. Terminal part numbers range from TERMINAL-FIRST-NUM to TERMINAL-LAST-NUM.

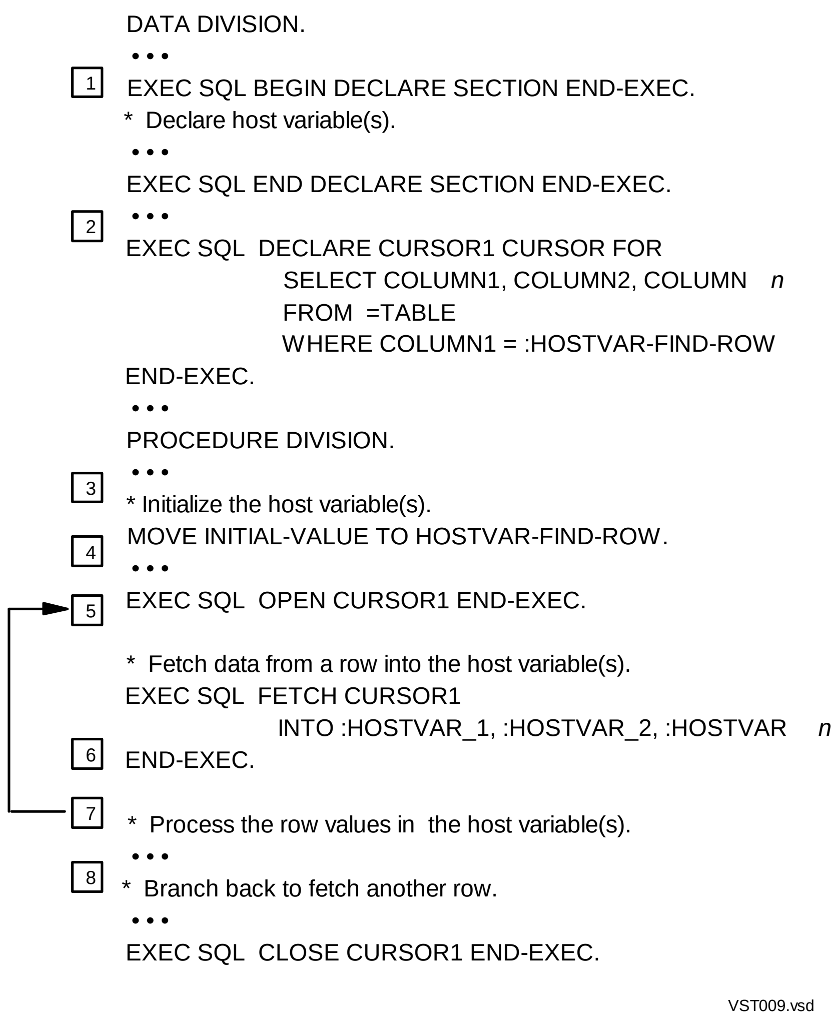
```
EXEC SQL DELETE FROM INVENT.PARTSUPP
     WHERE PARTNUM BETWEEN :TERMINAL-FIRST-NUM
                       AND :TERMINAL-LAST-NUM
     AND PARTCOST > :TERMINAL-MAX-COST  END-EXEC.
```

Using SQL Cursors

An SQL cursor is a named pointer that a host language program (C, COBOL, Pascal, or TAL) can use to access a set of rows in a table or view, one row at a time. Using a cursor, a program can process rows in the same way it might process records in a sequential file. The program can test the data in each row at the current cursor position and then, if the data meets certain criteria, the program can display, update, delete, or ignore the row.

Figure 4-1 shows the steps you follow to declare and use a static SQL cursor in a COBOL program. A cursor operation must execute each statement in this specified order. All steps are required, even if you execute the FETCH statement only once to retrieve a single row.

Figure 4-1. Using a Static SQL Cursor in a COBOL Program



```
      DATA DIVISION.
      • • •
 1    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     *  Declare host variable(s).
      • • •
      EXEC SQL END DECLARE SECTION END-EXEC.
      • • •
 2    EXEC SQL  DECLARE CURSOR1 CURSOR FOR
                    SELECT COLUMN1, COLUMN2, COLUMN   n
                    FROM  =TABLE
                    WHERE COLUMN1 = :HOSTVAR-FIND-ROW
      END-EXEC.
      • • •
      PROCEDURE DIVISION.
      • • •
 3    * Initialize the host variable(s).
 4    MOVE INITIAL-VALUE TO HOSTVAR-FIND-ROW.
      • • •
 5    EXEC SQL  OPEN CURSOR1 END-EXEC.

      *  Fetch data from a row into the host variable(s).
      EXEC SQL  FETCH CURSOR1
                   INTO :HOSTVAR_1, :HOSTVAR_2, :HOSTVAR   n
 6    END-EXEC.

 7    *  Process the row values in  the host variable(s).
      • • •
 8   *  Branch back to fetch another row.
      • • •
      EXEC SQL  CLOSE CURSOR1 END-EXEC.
```

Steps for Using a Cursor

These steps are shown in Figure 4-1 on page 4-12. Each step is described in detail on subsequent pages in this section.

1. Declare any host variables you plan to use with the cursor.
2. Name and define the cursor by using a DECLARE CURSOR statement. Follow the conventions for an SQL identifier for the cursor name. The DECLARE CURSOR statement also associates the cursor with a SELECT statement that specifies the rows to retrieve.
3. Initialize any host variables you specified in the WHERE clause of the SELECT statement in the DECLARE CURSOR statement.
4. Open the cursor by using an OPEN statement. The OPEN statement determines the result table and sorts the table if the SELECT statement includes the ORDER BY clause. For audited tables or views, the OPEN statement also associates the cursor with a TMF transaction.
5. Retrieve the column values from a row using the FETCH statement. The FETCH statement positions the cursor at the next row of the result table and transfers the column values defined in the associated SELECT statement to the corresponding host variables. The FETCH statement also locks each row according to the access specified by the SELECT statement.

   For audited tables or views, the FETCH statement must execute within the same TMF transaction as the OPEN statement.
6. Process the column values returned from the current row to the host variables. For example, you might test a value and then delete or update the row.
7. After you process the current row, branch back to the FETCH statement and retrieve the next row. Continue executing this loop until you have processed all rows specified by the associated SELECT statement (and SQLCODE equals 100).
8. Close the cursor using the CLOSE statement. The CLOSE statement releases the result table established by the OPEN statement. (The FREE RESOURCES statement also releases the result table.)

Access Requirements for Cursors

To use an SQL cursor, a process started by the program must have the access authority. SQL/MP checks this authority when the program opens the cursor.

Access	**SQL Objects**
Read	Tables or protection views referred to in the SELECT statement associated with the cursor (that is, specified in the DECLARE CURSOR statement)
Read	Tables or protection views underlying the shorthand view, if the cursor refers to a shorthand view
Write	Tables referenced, if the cursor declaration includes the FOR UPDATE clause

A program can use a cursor whose declaration does not specify FOR UPDATE to locate rows in a table to delete. SQL/MP tests the table only for read access when the OPEN statement executes. However, because a DELETE operation requires write access, SQL/MP checks for write access when you execute the DELETE statement.

A program contending for data access with other users can specify the IN EXCLUSIVE MODE clause in the associated SELECT statement. SQL/MP then does not have to convert the lock for a subsequent UPDATE or DELETE operation. However, if a program is reading records accessed concurrently by a cursor defined with an IN EXCLUSIVE MODE clause in another program, the first program must wait to access the data.

For information about process access, see Required Access Authority on page 7-1.

Cursor Position

The cursor position is similar to the record position in a sequential file. The SQL statements that affect the cursor position in a program are:

SQL Statement	**Cursor Position or Action**
OPEN	Positions the cursor before the first row.
FETCH	Positions the cursor at the retrieved row (or the current position).
DELETE	Positions the cursor between rows. For example, if the current row is deleted, the cursor is positioned either between rows or before the next row and after the preceding row.
SELECT	Determines the order in which the rows are returned. To specify an order, include an ORDER BY clause. Otherwise, the order is undefined.
CLOSE	Causes no position; release the result table established by the cursor.

Cursor Stability

Cursor stability guarantees that a row at the current cursor position cannot be modified by another program. For SQL/MP to guarantee cursor stability, declare the cursor with the FOR UPDATE clause or specify the STABLE ACCESS option. In some cases, a program might be accessing a copy of a row instead of the actual row. For example, a program might be accessing a copy of the row if the associated SELECT statement defining the cursor requires that the system perform any of these operations:

- Ordering the rows by a column
- Removing duplicate rows
- Performing other operations that require the selected table to be copied into a result table before it is used by a program

If your program is accessing a copy of a row instead of the actual row, the cursor points to a copy of the data, and the data is concurrently available to other programs. Accessing a copy of the data, however, never occurs if the cursor is declared with the

FOR UPDATE clause. In this case, your cursor points to the actual data and has cursor stability.

Virtual Sequential Block Buffering (VSBB)

The SQL/MP optimizer often uses Virtual Sequential Block Buffering (VSBB) as an access path strategy. Conflicting UPDATE, DELETE, or INSERT statements can invalidate a cursor’s buffering for a table. Each invalidation forces the next FETCH statement to send a message to the disk process to retrieve a new buffer, which can substantially degrade a program’s performance. These statements invalidate the buffer for cursor operations:

- An INSERT statement on the same table by the current process
- A stand-alone UPDATE or DELETE statement on the same table (directly or through a view) by the same process
- An UPDATE...WHERE CURRENT or DELETE...WHERE CURRENT statement using a different cursor to access the same table (directly or through a view) by the same process

For example, a loop containing both a FETCH statement and a stand-alone UPDATE or DELETE statement on the same table invalidates the cursor’s buffer on every loop iteration. You can minimize or eliminate this problem by following these guidelines:

- Do not use INSERT statements within a cursor operation.
- Use the UPDATE...WHERE CURRENT or DELETE...WHERE CURRENT statement for a cursor rather than a stand-alone UPDATE or DELETE statement.
- Do not open multiple cursors on a table if any of the cursors are used to update that table.

DECLARE CURSOR Statement

The DECLARE CURSOR statement names and defines a cursor and associates the cursor with a SELECT statement that specifies the rows to retrieve. A COBOL program requires no special authorization to execute a DECLARE CURSOR statement.

Follow these guidelines when you use a DECLARE CURSOR statement:

- The cursor name specified in the DECLARE CURSOR statement is an SQL identifier and is not case-sensitive. For example, SQL/MP considers Cur, cur, CUR, and CuR as equivalent names.
- Declare all host variables you use in the associated SELECT statement before the DECLARE CURSOR statement. Host variables must also be within the same scope as all SQL statements that refer to them.
- Place the DECLARE CURSOR statement in listing order before the other SQL statements, including the OPEN, FETCH, INSERT, DELETE, UPDATE, and

CLOSE statements, that refer to the cursor. The DECLARE CURSOR statement must also be within the scope of the statements that refer to the cursor.

- The DECLARE CURSOR statement does not affect the values in the SQLCA and SQLSA data structures.

Example 4-2 declares a cursor named LIST-BY-PARTNUM:

Example 4-2. Declaring a Cursor

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
01 PARTS.
   02 PARTNUM         PIC 9(4) DISPLAY.
   02 PARTDESC        PIC X(18).
   02 PRICE           PIC S9(16)V9(2) COMP.
   02 QTY-AVAILABLE   PIC S9(9) COMP.
...
EXEC SQL END DECLARE SECTION  END-EXEC.
...
EXEC SQL DECLARE LIST-BY-PARTNUM CURSOR FOR
       SELECT PARTNUM,
              PARTDESC,
              PRICE,
              QTY-AVAILABLE
        FROM   =PARTS
        WHERE  PARTNUM>= :PARTNUM OF PARTS
        ORDER BY PARTNUM
        BROWSE ACCESS;

PROCEDURE DIVISION.
...
```

OPEN Statement

The OPEN statement opens an SQL cursor. The OPEN operation orders and defines the set of rows in the result table and then positions the cursor before the first row.

The OPEN statement does not acquire any locks unless a sort is necessary to order the selected rows. (The FETCH statement acquires any locks associated with a cursor.)

To execute an OPEN statement for a cursor, a process started by the program must have the access authority described in Access Requirements for Cursors on page 4-13.

If the associated SELECT statement contains host variables in the WHERE clause, you must initialize these host variables before you execute the OPEN statement. When the OPEN statement executes, SQL/MP defines the set of rows in the result table and places the input host variables in its buffers. If you do not initialize the host variables before you execute the OPEN statement, these problems can occur:

- If a host variable contains values with unexpected data types, overflow or truncation errors can occur.

- If a host variable contains old values from the previous execution of the program, a subsequent FETCH statement uses these old values as the starting point to retrieve data. Therefore, the FETCH does not begin at the expected location in the result table.

The host variables must also be declared within the scope of the OPEN statement.

Some additional considerations for the OPEN statement are:

- You must code an OPEN statement within the scope of all other SQL statements (including the DECLARE CURSOR, FETCH, INSERT, DELETE, UPDATE, and CLOSE statements) that use the cursor.
- The OPEN statement must execute before any FETCH statements for the cursor.
- For audited tables and views, the OPEN statement must execute within a TMF transaction.
- If data is materialized by the OPEN operation, SQL/MP returns statistics to the SQLSA structure. For information about returning statistics to a program, see Section 9, Error and Status Reporting.
- If the DECLARE CURSOR statement for the cursor specifies a sort operation (for example, with an ORDER BY clause), do not issue an AWAITIO or AWAITIOX statement with the *filenum* parameter set to -1 after you open the cursor. Otherwise, the sort operation fails with SQL error -8301.

This OPEN statement opens the LIST-BY-PARTNUM cursor:

```
EXEC SQL OPEN LIST-BY-PARTNUM END-EXEC.
```

FETCH Statement

The FETCH statement positions the cursor at the next row of the result table and transfers a value from each column in the row specified by the associated SELECT statement to the corresponding host variable.

To execute a FETCH statement, a process started by the program must have read access to tables or views associated with the cursor. For information about process access, see Required Access Authority on page 7-1.

SQL/MP returns these values to SQLCODE after a FETCH statement.

SQLCODE Value	Description
`0`	The FETCH statement was successful.
`100`	The end of a table was encountered.
`< 0`	An error occurred; SQLCODE contains the error number.
`> 0 (not 100)`	A warning occurred; SQLCODE contains the first warning number.

The cursor must be open when the FETCH statement executes. The FETCH statement must also execute within the scope of all other SQL statements, including the DECLARE CURSOR, OPEN, INSERT, DELETE, UPDATE, and CLOSE statements that refer to the cursor.

SQL/MP resets values in an SQLSA structure immediately before a FETCH statement executes. If you use an SQLSA value elsewhere in your program, save the value in a variable immediately after the FETCH statement executes. To monitor statistics for a cursor, declare accumulator variables for the required values and add the SQLSA values to the accumulator variables after each FETCH statement executes.

For audited tables and views, the FETCH statement must execute within the same TMF transaction as the OPEN statement for the cursor.

This FETCH statement retrieves information from the PARTS table:

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
  01 PARTS.
     02 PARTNUM         PIC 9(4) DISPLAY.
     02 PARTDESC        PIC X(18).
     02 PRICE           PIC S9(16)V9(2) COMP.
     02 QTY-AVAILABLE   PIC S9(9) COMP.
...
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL DECLARE LIST-BY-PARTNUM CURSOR FOR
         SELECT PARTNUM,
         PARTDESC,
         PRICE,
         QTY-AVAILABLE
         FROM   =PARTS
         WHERE  PARTNUM  >= :PARTNUM OF PARTS
 ORDER BY PARTNUM
 BROWSE ACCESS END-EXEC.
...

...
PROCEDURE DIVISION.
...
0100-GET-DATA.

EXEC SQL OPEN LIST-BY-PARTNUM END-EXEC.

PERFORM 0200-FETCH-ROWS WITH TEST AFTER UNTIL
   SQLCODE OF SQLCA NOT = 0.
...
EXEC SQL CLOSE LIST-BY-PARTNUM END-EXEC.
...
0200-FETCH-ROWS.
EXEC SQL
  FETCH LIST-BY-PARTNUM
        INTO  :PARTNUM OF PARTS,
              :PARTDESC OF PARTS,
              :PRICE OF PARTS,
              :QTY-AVAILABLE OF PARTS
END-EXEC.
```

```
* Process the retrieved values in the host variables.
...
```

Multirow SELECT Statement

When used with a cursor, a SELECT statement can return multiple rows from a table or protection view, one row at a time. A cursor uses a FETCH statement to retrieve each row and store the selected column values in host variables. The program can then process the values (for example, list or save them in an array).

To execute a SELECT statement, a process started by a program must have read access to all tables, protection views, and the underlying tables of shorthand views used in the statement. For information about process access, see Required Access Authority on page 7-1.

All statements that refer to the cursor, including the DECLARE CURSOR, OPEN, FETCH, and CLOSE statements, must be within the same scope.

This example uses the GET-NAME-ADDRESS cursor to return the name and address of all customers within a certain range from the CUSTOMER table. For data consistency, the SELECT statement includes the REPEATABLE ACCESS clause to lock the rows. The BETWEEN clause specifies the range of zip codes, and the ORDER BY clause sorts the rows by zip code (POSTCODE).

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 HOST-VARIABLES.
   02 BEGIN-CODE     PIC X(10)
   02 END-CODE       PIC X(10)
 ...
 EXEC SQL INVOKE =CUSTOMER AS CUSTOMER-REC END-EXEC.
 ...
 EXEC SQL END DECLARE SECTION END-EXEC.

 EXEC SQL DECLARE GET-NAME-ADDRESS CURSOR FOR
   SELECT CUSTNAME, STREET, CITY, STATE, POSTCODE
   FROM   =CUSTOMER
   WHERE  POSTCODE BETWEEN :BEGIN-CODE AND :END-CODE
   ORDER BY POSTCODE
   REPEATABLE ACCESS END-EXEC.
 ...
 PROCEDURE DIVISION.
 ...
 EXEC SQL OPEN GET-NAME-ADDRESS END-EXEC.
* Set values for BEGIN-CODE and END-CODE.
 ...
 1000-FETCH-A-ROW.
 EXEC SQL FETCH GET-NAME-ADDRESS
              INTO :CUSTNAME OF CUSTOMER-REC,
                   :STREET OF CUSTOMER-REC,
                   :CITY OF CUSTOMER-REC,
                   :STATE OF CUSTOMER-REC
                   :POSTCODE OF CUSTOMER-REC
 END-EXEC.
```

```
* Process row values returned to host variables.
 ...
 EXEC SQL CLOSE GET-NAME-ADDRESS END-EXEC.
```

UPDATE Statement

When used with a cursor, an UPDATE statement updates rows, one row at a time, in a table or protection view. To identify the set of rows to update (or test), specify the FOR UPDATE OF clause in the associated SELECT statement. Before you update each row, you can test one or more column values. If you decide to update the row, specify the WHERE CURRENT OF clause in the UPDATE statement.

To execute an UPDATE statement, a process started by the program must have read and write access to the table or view being updated. It must also have read access to tables or views specified in subqueries of the search condition. For information about process access, see Required Access Authority on page 7-1.

Do not use a stand-alone UPDATE statement to update a row that has been retrieved using a FETCH statement. A stand-alone UPDATE statement invalidates the cursor's buffering for the table and can substantially degrade performance.

An UPDATE statement must be within the scope of all other SQL statements, including the DECLARE CURSOR, OPEN, FETCH, INSERT, and CLOSE statements, that refer to the cursor. For audited tables and views, the UPDATE statement must execute within the same TMF transaction as the OPEN and FETCH statements for the cursor.

Example 4-3 on page 4-21 uses the GET-BY-PARTNUM cursor and the host variables named NEW-PARTDESC, NEW-PRICE, and NEW-QTY to update the PARTS table.

This example also uses a cursor. Suppose that you want a cursor to position in the PARTS table on the part number specified by host variable STARTING-PARTNUM, so that the program can fetch rows and determine whether to update data in the columns. The row updated is at the current position of the cursor GET-BY-PARTNUM. The example declares the host variables NEW-PARTDESC, NEW-PRICE, and NEW-QTY and sets them to the new values for the columns before executing the UPDATE statement.

Example 4-3. Using the UPDATE statement

```
BEGIN DECLARE SECTION.
01 NEW-PARTS.
  02 STARTING-PARTNUM    PIC 9(4).
  02 NEW-PARTDESC        PIC X(18).
  02 NEW-PRICE           PIC S9(16)V9(2) COMP.
  02 NEW-QTY             PIC S9(9) COMP.
...
END DECLARE SECTION.

EXEC SQL DECLARE GET-BY-PARTNUM CURSOR FOR
         SELECT PARTNUM,
                PARTDESC,
                PRICE,
                QTY-AVAILABLE
         FROM   SALES.PARTS
         WHERE ( PARTNUM >= :STARTING-PARTNUM )
         STABLE ACCESS
         FOR UPDATE OF PARTDESC, PRICE, QTY-AVAILABLE
END-EXEC.
...
...
PROCEDURE DIVISION.
...
3000-UPDATE-PARTNUM.

MOVE FIRST-NUMBER TO STARTING-PARTNUM OF NEW-PARTS.

EXEC SQL OPEN GET-BY-PARTNUM END-EXEC.

* Fetch one row from the PARTS table.
EXEC SQL FETCH GET-BY-PARTNUM END-EXEC.

* Determine whether this is a row to be updated.
...
* If the row is to be updated, assign new values
* to NEW-PARTDESC, NEW-PRICE, and NEW-QTY.
* Update the row at the current cursor position.

EXEC SQL UPDATE SALES.PARTS
  SET PARTDESC      = :NEW-PARTDESC OF NEW-PARTS,
      PRICE         = :NEW-PRICE OF NEW-PARTS,
      QTY-AVAILABLE = :NEW-QTY OF NEW-PARTS
  WHERE CURRENT OF GET-BY-PARTNUM
END-EXEC.

* Branch back to fetch another row from the PARTS table.
...

EXEC SQL CLOSE GET-BY-PARTNUM END-EXEC.
```

Multirow DELETE Statement

When used with a cursor, a DELETE statement deletes multiple rows one row at a time from a table or protection view. You identify the set of rows to delete (or test) in the associated SELECT statement. Before you delete a row, you can test one or more column values, and then, if you decide to delete the row, specify the WHERE CURRENT OF clause in the DELETE statement.

If you delete all rows from a table, the table still exists until it is deleted from the catalog with a DROP TABLE statement.

To execute a DELETE statement, a process started by a program must have read and write access to the table or view containing the rows to be deleted and to tables or views in subqueries of the search condition. For information about process access, see Required Access Authority on page 7-1.

A DELETE statement must execute within the scope of all other SQL statements, including the DECLARE CURSOR, OPEN, FETCH, INSERT, and CLOSE statements, that refer to the cursor. For audited tables and views, the DELETE statement must execute within the same TMF transaction as the OPEN and FETCH statements for the cursor.

Note. Do not use a stand-alone DELETE statement to delete a row that has been retrieved using a FETCH statement. A stand-alone DELETE statement can invalidate the cursor's buffering for the table and degrade performance.

This example declares a cursor named GET-BY-PARTNUM, fetches data from the PARTS table, tests the data, and then deletes specific rows:

```
 EXEC SQL DECLARE GET-BY-PARTNUM CURSOR FOR
          SELECT PARTNUM,
                 PARTDESC,
                 PRICE,
                 QTY-AVAILABLE
          FROM SALES.PARTS
          WHERE (PARTNUM >= :PARTNUM OF PARTS)
          ORDER BY PARTNUM
          STABLE ACCESS END-EXEC.

 PROCEDURE DIVISION.
 ...
 EXEC SQL OPEN GET-BY-PARTNUM END-EXEC.

 EXEC SQL FETCH GET-BY-PARTNUM  ...  END-EXEC.

* Test the value(s) in the current row.
 ...
* Delete the current row.
 EXEC SQL DELETE FROM SALES.PARTS
          WHERE CURRENT OF GET-BY-PARTNUM
 END-EXEC.
 ...
 EXEC SQL CLOSE GET-BY-PARTNUM END-EXEC.
```

CLOSE Statement

The CLOSE statement closes an open SQL cursor. After the CLOSE statement executes, the result table established by the OPEN statement no longer exists. To use the cursor again, you must reopen it using an OPEN statement.

A program does not require special authorization to execute a CLOSE statement.

A CLOSE statement must be within the scope of all other SQL statements, including the DECLARE CURSOR, OPEN, FETCH, INSERT, DELETE, and UPDATE statements, that refer to the cursor.

This CLOSE statement closes the LIST-BY-PARTNUM cursor:

```
...
EXEC SQL CLOSE LIST-BY-PARTNUM END-EXEC.
```

Only an explicit CLOSE statement (or a FREE RESOURCES statement) closes an open SQL cursor. The CLOSE operation releases the resources used by the cursor and frees any locks the cursor holds. If you are planning to reuse a cursor later in your program, you can usually leave it open to save the overhead of opening it. However, if your program is a Pathway server, always close an open cursor before returning control to the requester, especially if the requester initiated a TMF transaction.

Using Foreign Cursors

Foreign cursors are cursors that are not declared in the program or procedure in which they are referenced. Foreign cursors can be static or dynamic.

A reference to a foreign cursor contains two parts, a procedure name part and a cursor name part. This example refers to a foreign cursor named LIST-BY-PARTNUM which is declared in the procedure 3000-UPDATE-INVENTORY:

```
3000-UPDATE-INVENTORY.LIST-BY-PARTNUM
```

A foreign cursor reference can appear in an OPEN, FETCH, or CLOSE cursor statement. It references a cursor that is declared in another procedure, which is not necessarily in the same source file. References to a dynamic foreign cursor are resolved at run time by the SQL executor.

The prepare and dynamic cursor declarations must be in the same procedure so that the resolution between the prepare and the cursor declaration can occur to detect whether a statement name has been prepared or not, and to maintain proper association between a procedure and a particular statement name.

These statements open, fetch, and close a foreign cursor named `LIST-BY-PARTNUM` which is declared in the procedure `3000-UPDATE-INVENTORY`:

```
** While EOF=false

OPEN 3000-UPDATE-INVENTORY.LIST-BY-PARTNUM USING DESCRIPTOR
input-sqlda

FETCH 3000-UPDATE-INVENTORY.LIST-BY-PARTNUM USING DESCRIPTOR
output-sqlda.

CLOSE 3000-UPDATE-INVENTORY.LIST-BY-PARTNUM.
```

5 SQL/MP System Procedures

Table 5-1 describes the NonStop SQL/MP system procedures a COBOL program can call to return various SQL information. These procedures are described alphabetically on subsequent pages in this section.

Table 5-1. SQL/MP System Procedures

Procedure	Description
To Use With Dynamic SQL Operations	
SQLADDR on page 5-3	Returns the address of an input parameter, output variable, or indicator variable to an input or output SQLDA structure (used only in dynamic SQL operations)
To Return Error and Warning Information	
SQLCA_DISPLAY2_ on page 5-4	Writes to a file or terminal the error and warning messages that SQL/MP returns to the SQLCA structure
SQLCA_TOBUFFER2_ on page 5-11	Returns to a record area in the program the error or warning messages that SQL/MP returns to the SQLCA structure
SQLCAFSCODE on page 5-17	Returns information about file-system, disk-process, or operating system errors from the SQLCA structure
SQLCAGETINFOLIST on page 5-17	Returns to an area in the program a specified subset of the error or warning information in the SQLCA structure
To Return Version Information	
SQLGETCATALOGVERSION on page 5-24	Returns the version of an SQL catalog
SQLGETOBJECTVERSION on page 5-25	Returns the version of an SQL object (table, index, or view)
SQLGETSYSTEMVERSION on page 5-26	Returns the version of the SQL file-system and disk-process components for a specified system
To Return Execution Statistics	
SQLSADISPLAY on page 5-27	Writes to a file or terminal the execution statistics that SQL/MP returns to the SQLSA structure
To Return Error and Warning Information (Superseded Procedures on page 5-30)	
SQLCADISPLAY on page 5-30	Writes error or warning messages from the SQLCA structure to a file or terminal (superseded by SQLCA_DISPLAY2_)
SQLCATOBUFFER on page 5-34	Writes error or warning messages from the SQLCA structure to a record area (superseded by SQLCA_TOBUFFER2_)

COBOLEXT File

To call the SQL/MP system procedures, which are written in TAL, use the COBOL ENTER TAL statement. The COBOLEXT file contains source declarations for these procedures (as well as for other system procedures). You might need to check with your system administrator to make sure the COBOLEXT file for the procedures you use in your program are available on your system. For more information about the COBOLEXT file and the ENTER TAL statement, see the *COBOL85 for NonStop Systems Manual*.

Guardian System Procedures

In addition to procedures in Table 5-1 on page 5-1, a COBOL program can also call the Guardian system procedures described in Table 5-2 to return information about SQL objects and programs. For a detailed description of these procedures, see the *Guardian Procedure Calls Reference Manual*.

Table 5-2. Guardian System Procedures that Return SQL Information

Procedure	Description
FILE_GETINFO_	Returns limited information, including the last error and type, about a file using the file number
FILE_GETINFOBYNAME_	Returns limited information about a file using the file number
FILE_GETINFOLIST_	Returns detailed information about a file using the file number. Item codes 40, 82, 83, 84, and 85 apply to SQL/MP
FILE_GETINFOLISTBYNAME_	Returns detailed information about a file using the file name. Item codes 40, 82, 83, 84, and 85 apply to SQL/MP

SQL Message File

The SQLMSG file contains error messages, informational messages, and help text used by SQLCI, the SQL compiler, and host-language programs. The default SQL message file is $SYSTEM.SYSTEM.SQLMSG. A COBOL program opens and reads the SQL message file when it calls an SQL system procedure that returns error or status information (for example, SQLCA_DISPLAY2_ or SQLCA_TOBUFFER2_).

The SQLMSG file contains text in English. You can specify a different SQL message file (for example, a file translated into French) with the =_SQL_MSG_*system* DEFINE. For the alternate SQL message files available on your node, ask your database administrator or service provider.

You can add (or modify) the =_SQL_MSG_*system* DEFINE either interactively from TACL or SQLCI or programmatically from a COBOL program:

- From TACL or SQLCI, enter an ADD DEFINE (or ALTER DEFINE) command. Do not include a backslash (\) or a space before the node name. For example, this command adds a new DEFINE for the $SQL.MSG.FRENCH message file on the \PARIS node:

  ```
  ADD DEFINE =_SQL_MSG_PARIS,CLASS MAP,FILE $SQL.MSG.FRENCH
  ```

  For the _SQL_MSG_*system* DEFINE to be in effect for an SQLCI session, you must add or change the DEFINE before you start the SQLCI session. If you add or change the DEFINE after you start the session, SQL/MP returns warning message 10201, which indicates that the DEFINE has been changed but the old message file is still in effect.

- From a COBOL program, call the DEFINEADD (or DEFINESETATTR) system procedure. Your program must add or alter the DEFINE before it calls a system procedure that opens and reads the SQL message file. Otherwise, your program uses the default message file. For more information about system procedures, see the *Guardian Procedure Calls Reference Manual*.

SQLADDR

The SQLADDR procedure returns the address of an input parameter, output variable, or indicator variable to an input or output SQLDA structure. This procedure is used only for dynamic SQL operations.

```
ENTER TAL "SQLADDR"
           USING variable-name
           GIVING { VAR-PTR }
                  { IND-PTR }
           OF SQLVAR OF sqlda-name.
```

variable-name

is the name of an input parameter, output variable, or indicator variable. *variable-name* must be defined in the Data Division.

Specify the name in the GIVING clause:

- If *variable-name* is the name of an input parameter or output variable, specify VAR-PTR in the GIVING clause.
- If *variable-name* is the name of an indicator variable, specify IND-PTR in the GIVING clause.

sqlda-name

is the name of an input or output SQLDA structure. *sqlda-name* is defined in the Data Division by an INCLUDE SQLDA directive.

For more information about the SQLADDR procedure, see Section 10, Dynamic SQL Operations.

SQLCA_DISPLAY2_

The SQLCA_DISPLAY2_ procedure displays the error or warning messages that SQL/MP returns to the SQLCA structure. SQLCA_DISPLAY2_ writes the information to a file or to a terminal.

The information returned to the SQLCA structure can originate from these subsystems or system components:

- SQL/MP
- NonStop OS
- File system
- Disk process (DP2)
- FastSort program (SORTPROG process)
- Sequential I/O (SIO) procedures

SQL/MP communicates errors, warnings, and statistics to a program through the SQLCA structure. However, because the SQLCA contains information in a format that is not appropriate to display, you must call the SQLCA_DISPLAY2_ procedure to convert this information to an appropriate format.

```
ENTER TAL "SQLCA_DISPLAY2_" USING
                               sqlca,
                             [ output-file-number,    ]
                             [ output-record-length,  ]
                             [ sql-msg-file-number,   ]
                             [ errors,                ]
                             [ warnings,              ]
                             [ statistics,            ]
                             [ caller-error-loc,      ]
                             [ internal-error-loc,    ]
                             [ prefix,                ]
                             [ prefix-length,         ]
                             [ suffix,                ]
                             [ suffix-length          ].
```

sqlca   required   input

is the record name of the SQLCA structure.

output-file-number   optional   input

`PIC S9(4) COMP`

is the output file number. If you omit this value or set it to a negative value, SQLCA_DISPLAY2_ displays information on your home terminal.

`output-record-length` optional input

`PIC S9(4) COMP`

is the length in bytes of records to be written to the output file. The length must be an integer value from 60 to 600.

The default length is 79 bytes.

`sql-msg-file-number` optional input/output

`PIC S9(4) COMP`

is the file number of the SQL message file (SQLMSG is the default file). If you specify -1 as the input value, the system opens the message file and returns the resulting file number. If you specify a value other than -1, the system uses that value as the file number of the message file.

To improve the performance of multiple calls to the SQLCA_DISPLAY2_ procedure, specify -1 on the first call and then use the returned file number for subsequent calls. By using the file number, the system opens the file only once and uses the file number for subsequent calls. Otherwise, the system opens the file for each call.

`errors` optional input

`PIC X`

controls the display of error messages:

Y	Display all errors.
N	Display only the first error.
B	Display all errors but suppress this prefix: `ERROR from` *subsystem* `[`*nn*`]`

The default is Y.

`warnings` optional input

`PIC X`

controls the display of warning messages:

Y	Display all warning messages.
N	Do not display any warning messages.
B	Display all warnings but suppress this prefix: `WARNING from` *subsystem* `[`*nn*`]`

The default is Y.

`statistics` `optional` `input`

`PIC X`

controls the display of statistics:

Y	Display row and cost statistics if the value returned to the SQLCA in the ROW or COST field is greater than or equal to 0.
N	Do not display statistics.
R	Display row statistics only.
C	Display cost statistics only.

The default is Y.

`caller-error-loc` `optional` `input`

`PIC X`

controls the display of the program name and line number of the SQL statement that received the error:

Y	Display the program name and line number.
N	Suppress the display.

The default is Y.

`internal-error-loc` `optional` `input`

`PIC X`

controls the display of the system-code location where the first error in the SQLCA occurred:

Y	Display the location.
N	Suppress the display.

The default is Y.

`prefix` `optional` `input`

`PIC X(`*`length`*`)`

is a string that the program uses to precede each output line. The default is three asterisks and a space (***).

`prefix-length` `optional` `input`

`PIC S9(4) COMP`

is the length of the *`prefix`* string for each output line. This length must be an integer value from 1 to 15. If you include *`prefix`*, *`prefix-length`* is required.

`suffix`	`optional`	`input`

`PIC X(`*`length`*`)`

is a string to be appended to each output line. The default is a null string.

`suffix-length`	`optional`	`input`

`PIC S9(4) COMP`

is the length of the *`suffix`* string for each output line. This length must be an integer value from 1 to 15. If you include *`suffix`*, *`suffix-length`* is required.

Using SQLCA_DISPLAY2_ With an Error Table

If you plan to write the buffer to an SQL table for subsequent access through SQLCI, you might want to reduce the number of lines of error information and the amount of information in each line. To reduce the line length and the number of lines, specify SQLCA_DISPLAY2_ parameters that suppress the statistics and the internal error location and change the prefix to a single space.

Making these changes results in two or three lines per error at the most, or a maximum of 14 to 21 lines for the rare case where seven errors are returned. In most cases, space for four errors is sufficient. If you set the line length to 80 characters, four errors require a buffer of 960 characters.

If you use SQLCA_TOBUFFER2_ to write to an SQL error table, make the same changes to the parameter defaults.

When you create the table to receive the error information, specify the text columns as multiples of 80 but less than 255 characters each. If you use SQLCI to retrieve error information from an error table, it displays a maximum of 255 characters per column. If you put the entire buffer in one column, SQLCI displays only the first 255 characters of text. To avoid truncation and allow 80-character lines, define text columns of 240 bytes each. Depending on the size of the buffer, you might need two, three, or four columns to hold error information.

Additional Considerations for SQLCA_DISPLAY2_

- SQL/MP returns errors as negative numbers and warnings as positive numbers. Therefore, you might need to modify your program accordingly.
- If there is no text for an error number, SQL/MP displays:

  ```
  No error text found.
  ```

  If you receive this message, the version of the SQL message file might be invalid. To determine the version of the SQL message file, use the SQLCI ENV command and check the version specified by MESSAGEFILEVSRN.
- If the error text exceeds *`output-record-length`*, the output is folded at word boundaries, which produces subsequent lines indented five spaces.

- The SQLCA can contain a maximum of seven errors and 180 bytes of text of the actual parameters returned to the program. Information that exceeds these limits is lost. SQLCA_DISPLAY2_ displays a warning message that indicates when information is lost.

Example 5-1 on page 5-9 shows a program that performs these functions:

- Processes these two transactions in a single server:
    - TRANS-CODE-1 retrieves a row from a table and displays the row values on a terminal screen.
    - TRANS-CODE-2 updates the previously retrieved row with data entered at the terminal.
- Processes these reply codes for the transactions:

REPLY-CODE = 0000	Successful operation occurred.
REPLY-CODE = 9998	Operation failed; record not found.
REPLY-CODE = 9999	Error, operation failed; backout transaction.

- Displays an advisory message for each transaction to indicate the result of the requested operation. The message must be displayed in the advisory line (line 25) of the terminal screen and be self-explanatory, because the terminal operator has no knowledge of SQL and no access to an SQL error messages manual.
- Processes any error or warning conditions:
    - Sends SQL warnings only to HOMETERM and not to the terminal. SQL warnings occur rarely and usually have no meaning for a terminal operator.
    - Routes SQL errors to HOMETERM for analysis by the database administrator, suppressing statistics and the internal location of the error message.

  HOMETERM is the terminal used by subsystems such as Pathway and TMF to receive error messages that must be processed by an operator or database administrator.

Example 5-1. Error Processing Using SQLCA_DISPLAY2_

```
WORKING-STORAGE SECTION.
 EXEC SQL INCLUDE SQLCA END-EXEC.
 PROCEDURE DIVISION.
 EXEC SQL WHENEVER NOT FOUND PERFORM  :8000-NOT-FOUND END-EXEC.
 EXEC SQL WHENEVER SQLWARNING PERFORM :9900-SQL-WARN  END-EXEC.
 EXEC SQL WHENEVER SQLERROR PERFORM   :9999-SQL-ERROR END-EXEC.

 DO-BEGIN.
    PERFORM READ-MESSAGE.
    IF NOT EOF-MESSAGE-IN
      MOVE MSG-KEY TO HOSTVAR-KEY
      MOVE ZERO TO REPLY-CODE
      PERFORM DO-TRANS
      PERFORM WRITE-REPLY.

 DO-TRANS.
   EVALUATE TRANS-CODE OF INPUT MSG
   WHEN   1   PERFORM TRANS-CODE-1
   WHEN   2   PERFORM TRANS-CODE-2
   WHEN OTHER PERFORM INVALID-TRANS-CODE.

 TRANS-CODE-1.
 MOVE "REQUESTED ROW DISPLAYED" TO ADVISORY-LINE OF REPLY
 EXEC SQL SELECT COL1, COL2
            FROM TABLE
            INTO :HOSTVAR1, :HOSTVAR2
          WHERE KEY = :HOSTVAR-KEY
           FOR BROWSE ACCESS
 END-EXEC.

 MOVE HOSTVAR1 TO REPLY-FIELD1.
 MOVE HOSTVAR2 TO REPLY-FIELD2.
...
 TRANS-CODE-2.
 MOVE "SPECIFIED ROW UPDATED" TO ADVISORY-LINE OF REPLY.
 MOVE MSG-COL1 TO HOSTVAR1.
 MOVE MSG-COL2 TO HOSTVAR2.
 EXEC SQL UPDATE TABLE
             SET COL1 = :HOSTVAR1,
                 COL2 = :HOSTVAR2
          WHERE KEY = :HOSTVAR-KEY
 END-EXEC.
...
*******************************************************************
*  Copy error processing routines from the copy library COPYCODE
*******************************************************************

COPY   REQUEST-NOT-FOUND  OF COPYCODE.
COPY   SQL-WARNING        OF COPYCODE.
COPY   SQL-ERROR          OF COPYCODE.
```

Example 5-2 shows the routines in the file COPYCODE.

Example 5-2. Error Routines in Copy Library

```
?SECTION REQUEST-NOT-FOUND, TANDEM
8000-NOT-FOUND.
  MOVE "REQUESTED ROW NOT FOUND" TO ADVISORY-LINE OF REPLY.
   MOVE 9998 TO REPLY-CODE.

?SECTION SQL-WARNING, TANDEM
9900-SQL-WARN.
  MOVE -1 to SQL-MSG-FILE-NO.
  MOVE "N" to STATS.
  MOVE "N" to ERR-LOC.
*****************************************************************
* Send the SQL Warning message to HOMETERM for DBA analysis.      *
*****************************************************************
  ENTER TAL "SQLCA_DISPLAY2_" USING SQLCA,
                               OMITTED,
                               OMITTED,
                               SQL-MSG-FILE-NO,
                               OMITTED,
                               OMITTED,
                               STATS,
                               OMITTED,
                               ERR-LOC.
?SECTION SQL-ERROR, TANDEM
9999-SQL-ERROR.
  MOVE "PROCESS ERROR SQL =     FS =   - NOTIFY DB ADMINISTRATOR"
           TO ADVISORY-LINE OF REPLY.
  MOVE 9999 TO REPLY-CODE.
*****************************************************************
*                                                               *
*  Move the SQL return code to the advisory line and multiply it  *
*     by -1 to show a positive number.                          *
*  Move any file system error to the advisory line.             *
*                                                               *
*****************************************************************
  MOVE SQLCODE OF SQLCA TO ADVISORY-SQL OF ADVISORY-LINE.
  MULTIPLY ADVISORY-SQL OF ADVISORY LINE BY -1
       GIVING ADVISORY-SQL OF ADVISORY-LINE END-MULTIPLY.

  ENTER TAL "SQLCAFSCODE" USING SQLCA GIVING MY-FS-CODE.
  MOVE MY-FS-CODE TO ADVISORY-FS OF ADVISORY-LINE.

  MOVE -1 to SQL-MSG-FILE-NO.
  MOVE "N" to STATS.
  MOVE "N" to ERR-LOC.

*****************************************************************
* Send the SQL error message to HOMETERM for DBA analysis.        *
*****************************************************************
  ENTER TAL "SQLCA_DISPLAY2_" USING SQLCA,
                               OMITTED,
                               OMITTED,
                               SQL-MSG-FILE-NO,
                               OMITTED,
                               OMITTED,
                               STATS,
                               OMITTED,
                               ERR-LOC.
```

As shown in Example 5-2 on page 5-10, the messages sent to the HOMETERM contain neither the internal location where the error was encountered nor the statistics and cost of the SQL statement. Usually, this information is not important and can be omitted to reduce the number of error lines.

Consider an SQL constraint that is violated during the update, causing this message to be sent to HOMETERM:

```
SQLCA display of SQL statement at SAMPLE.#5141.854 process \AAA.$BBB
ERROR from SQL [-8233] Constraint Number 2 violated on base table T1.
```

Upon receiving this message, the database administrator can query the catalog through SQLCI to display the actual constraint predicate that was violated. This information is in the catalog CONSTRNT table. For this particular error, it is the second entry for the table T1.

Generating Meaningful Messages

The previous error message illustrates why it is sometimes difficult for a program to generate meaningful messages for display at a terminal. For example, consider the steps required for a program to generate a message such as “DATA FOR *colname* EXCEEDS LIMITS” that identifies the specific column in error:

1. The error routine might call SQLCA_TOBUFFER2_ (instead of SQLCA_DISPLAY2_) so the program can examine the buffer to retrieve the SQL error message with the constraint number and the table name.
2. If the program has read access to the catalog tables, it could develop a query against the CONSTRNT table by using the constraint number and table name returned by SQLCA_TOBUFFER2_ to retrieve more information about the constraint.
3. The program could then generate a message using the constraint information from the CONSTRNT table and return this message to the terminal.

SQLCA_TOBUFFER2_

The SQLCA_TOBUFFER2_ procedure writes to a buffer the error or warning messages that SQL/MP returns to the application program. The buffer is a record area declared in the Working-Storage or Extended-Storage Section.

The information returned to the buffer can originate from these subsystems or system components:

- SQL/MP
- NonStop OS
- File system
- Disk process (DP2)
- FastSort program (SORTPROG process)
- Sequential I/O (SIO) procedures

This procedure is similar to the SQLCA_DISPLAY2_ procedure, which writes error information to a file or terminal.

```
ENTER TAL "SQLCA_TOBUFFER2_" USING
                             sqlca,
                             output-buffer,
                             output-buffer-length,
                           [ first-record-number,  ]
                           [ output-records,       ]
                           [ more,                 ]
                           [ output-record-length, ]
                           [ sql-msg-file-number,  ]
                           [ errors,               ]
                           [ warnings,             ]
                           [ statistics,           ]
                           [ caller-error-loc      ]
                           [ internal-error-loc,   ]
                           [ prefix,               ]
                           [ prefix-length,        ]
                           [ suffix,               ]
                           [ suffix-length         ].
```

`sqlca                         required         input`

is the record name of the SQLCA.

The SQLCA is declared automatically when you give the INCLUDE SQLCA directive in the Working-Storage Section.

`output-buffer                 required         input/output`

`PIC X(length)`

is the record name to which SQLCA_TOBUFFER2_ writes the error information.

`output-buffer-length          required         input`

`PIC S9(4) COMP`

is the length of *output-buffer* in bytes. The length must be:

- An integer value from *output-record-length* through 600
- A multiple of *output-record-length*.

The minimum length recommended is 300 bytes.

`first-record-number           optional         input`

`PIC S9(4) COMP`

is the ordinal number of the first error record (line) to be moved to the output buffer. The procedure discards any error records with a lower number. The count of lines moved begins with 1.

The default is 1.

To obtain more than one error record, you must increment the value in *`first-record-number`*.

`output-records`	`optional`	`output`

`PIC S9(4) COMP`

is the number of records (lines) written by SQLCA_TOBUFFER2_ to *`output-buffer`*.

`more`	`optional`	`output`

`PIC X`

is a flag that indicates whether all the desired lines fit into the *`output-buffer`*:

Y	There were additional records; the buffer overflowed.
N	There were no additional records.

`output-record-length`	`optional`	`input`

`PIC S9(4) COMP`

defines the length of records to be written to the *`output-buffer`*. The length must be an integer value from 60 to 600.

The default length is 79 bytes.

The procedure pads each line with spaces and adds the suffix and prefix strings if the ENTER statement specifies them.

`sql-msg-file-number`	`optional`	`input/output`

`PIC S9(4) COMP`

is the file number of the SQL message file (SQLMSG is the default file). If you specify -1 as an input value, the system opens the message file and returns the resulting file number. If you specify a value other than -1, the system uses that value as the file number of the message file.

To improve the performance of a program that makes multiple calls to the SQLCA_TOBUFFER2_ procedure, specify -1 on the first call and then use the returned file number for subsequent calls. By using the file number, the system opens the file only once and uses the file number for subsequent calls. Otherwise, the system opens the file for each call.

The SQLMSG file contains text in English. You can specify a different SQL message file with the =_SQL_MSG_*`system`* DEFINE. For more information, see SQL Message File on page 5-2.

`errors` optional input

`PIC X`

controls the writing of error messages to the buffer:

Y	Write all errors.
N	Write only the first error.
B	Write all errors but suppress this prefix: `ERROR from` *`subsystem`* `[`*`nn`*`]:`

The default is Y.

`warnings` optional input

`PIC X`

controls the writing of warning messages to the buffer:

Y	Write all warning messages.
N	Do not write any warning messages.
B	Write all warnings but suppress this prefix: `WARNING from` *`subsystem`* `[`*`nn`*`]`

The default is Y.

`statistics` optional input

`PIC X`

controls the writing of statistics to the buffer:

Y	Write row and cost statistics if the value returned to the SQLCA in the ROW or COST field is greater than or equal to 0.
N	Do not write statistics.
R	Write row statistics only.
C	Write cost statistics only.

The default is Y.

`caller-error-loc` optional input

`PIC X`

controls the writing of the program name and line number of the SQL statement that received the error:

Y	Write the program name and line number.
N	Suppress the information.

The default is Y.

`internal-error-loc`   `optional`   `input`

`PIC X`

controls the writing of the system-code location where the first error in the SQLCA occurred:

Y	Write the location.
N	Suppress the information.

The default is Y.

`prefix`   `optional`   `input`

`PIC X(length)`

is a string to precede each output line. The default is three asterisks and a space (***).

`prefix-length`   `optional`   `input`

`PIC S9(4) COMP`

is the length of the *prefix* string for each output line. This length must be an integer value from 1 to 15. If you include *prefix*, *prefix-length* is required.

`suffix`   `optional`   `input`

`PIC X(length)`

is a string to be appended to each output line. The default is a null string.

`suffix-length`   `optional`   `input`

`PIC S9(4) COMP`

is the length of the *suffix* string for each output line. This length must be an integer value from 1 to 15. If you include *suffix*, *suffix-length* is required.

Using SQLCA_TOBUFFER2_ With an Error Table

If you plan to write the buffer to an SQL table for subsequent access through SQLCI, you might want to reduce the number of lines of error information and the amount of information in each line. To reduce the line length and the number of lines, specify SQLCA_DISPLAY2_ parameters that suppress the statistics and the internal error location and change the prefix to a single space.

Making these changes results in two or three lines per error at the most, or a maximum of 14 to 21 lines for the rare case where seven errors are returned. In most cases, space for four errors is sufficient. If you set the line length to 80 characters, four errors require a buffer of 960 characters.

If you use SQLCA_DISPLAY2_ to write to an SQL error table, make the same changes to the parameter defaults.

When you create the table to receive the error information, specify the text columns as multiples of 80 but less than 255 characters each. If you use SQLCI to retrieve error information from an error table, it displays a maximum of 255 characters per column. If you put the entire buffer in one column, SQLCI displays only the first 255 characters of text. To avoid truncation and allow 80-character lines, define text columns of 240 bytes each. Depending on the size of the buffer, you might need two, three, or four columns to hold error information.

Additional Considerations for SQLCA_BUFFER2_

Additional considerations for the SQLCA_BUFFER2_ procedure are:

- SQL/MP returns errors as negative numbers and warnings as positive numbers. Therefore, you might need to modify your program accordingly.
- If there is no text for an error number, SQL/MP displays:

  ```
  No error text found
  ```

  If you receive this message, the version of the SQL message file might be invalid. To determine the version of the SQL message file, use the SQLCI ENV command and check the version specified by MESSAGEFILEVSRN.
- SQLCA_TOBUFFER2_ works by starting with the *first-record-number* indicated to move output lines to the record area until all error messages are moved or until the text fills the record area. SQLCA_TOBUFFER2_ returns to *output-records* a count of the lines moved to the buffer. If overflow occurs, the procedure sets the *more* flag to Y.
- On an overflow condition, the program can retrieve the remainder of the error message text by calling SQLCA_TOBUFFER2_ again, setting *first-record-number* to *output-records* + 1.

This example uses a 75-character output record and declares a buffer SQLMSG-BUFFER as 375 characters. The ENTER statement specifies the SQLMSG file number with data item SQLMSG-FILENUM set to -1. The statement returns the file number so that subsequent calls retain the number.

```
WORKING-STORAGE SECTION.

01  SQLMSG-BUFFER        PIC X(375).
01  SQLMSG-FILENUM       PIC S9(4) COMP   VALUE -1.

PROCEDURE DIVISION.
  ...
  ENTER TAL "SQLCA_TOBUFFER2_" USING
                    SQLCA,
```

```
                             SQLMSG-BUFFER,
                             375,
                             OMITTED,
                             OMITTED,
                             OMITTED,
                             75,
                             SQLMSG-FILENUM.
       ...
```

SQLCAFSCODE

The SQLCAFSCODE procedure returns either the first or the last error in the SQLCA structure that was set by the file system, disk process, or operating system. If there was no such error, SQLCAFSCODE returns 0. If the SQLCA is full when an error occurs, the error is lost.

```
ENTER TAL "SQLCAFSCODE" USING
                          sqlca,
                        [ first-flag ]
                        GIVING error-info.
```

sqlca required input

is the record name of the SQLCA, which is declared automatically when you include the INCLUDE SQLCA directive.

first-flag optional input

PIC S9(4) COMP

specifies whether the first or the last error is set in the SQLCA:

Nonzero value (or omitted)	First error
0 (zero)	Last error

The default is the first error.

error-info required output

PIC S9(4) COMP

specifies the error you are requesting. If no error is returned, *error-info* is 0.

SQLCAGETINFOLIST

The SQLCAGETINFOLIST procedure returns error or warning information that SQL/MP sets in the SQLCA structure. You specify a list of numbers, called item codes, to specify the error or warning information, and SQLCAGETINFOLIST returns the information to a buffer in your program.

The information in the SQLCA structure can originate from these subsystems or system components:

- SQL/MP
- NonStop OS
- File system
- Disk process (DP2)
- FastSort program (SORTPROG process)
- Sequential I/O (SIO) procedures

Note. The SQLCAGETINFOLIST procedure returns error numbers as positive values and warning numbers as negative values. A program might need to switch the sign before processing the error or warning.

```
ENTER TAL "SQLCAGETINFOLIST" USING
                               sqlca,
                               item-list,
                               number-items,
                               result,
                               result-max,
                             [ error-index, ]
                             [ names-max,   ]
                             [ params-max,  ]
                             [ result-len,  ]
                             [ error-item   ]
                               GIVING call-error.
```

`sqlca` required input

is the record name of the SQLCA structure.

`item-list` required input

`PIC X(length-of-table)`

is a table of item codes that describes the information you want returned in the `result` buffer. For a list of these codes, see call-error output on page 5-20.

`number-items` required input

`PIC S(9) COMP`

is the number of items you specified in the `item-list` table.

`result` required output

`PIC X(length-of-table)`

is a table you define to receive the requested information. The items are returned in the order you specified in `item-list`. Each item is aligned on a word boundary.

`result-max`	`required`	`input`

`PIC S9(4) COMP`

is the maximum size, in bytes, of the *`result`* table.

`error-index`	`optional`	`input`

`PIC S9(4) COMP`

is the index of the SQLCA error entry you want to see.

The SQLCA structure has a fixed set of fields (item codes 1 through 21) for errors and warnings. In addition, SQLCA has a table of records (item codes 22 through 29), with each record describing one error or warning. SQL/MP uses *`error-index`* to access this table to determine the error or warning.

If *`error-index`* is omitted, the first error record is returned.

`names-max`	`optional`	`input`

`PIC S9(4) COMP`

is the maximum length your program allows for procedure IDs or file names (item codes 9, 13, and 19). Names that exceed this length are truncated (no error results from the truncation).

`params-max`	`optional`	`input`

`PIC S9(4) COMP`

is the maximum length your program allows for parameter information (item codes 16 and 29). Parameter information that exceeds this length is truncated (no error results from the truncation).

`result-len`	`optional`	`output`

`PIC S9(4) COMP`

returns the total number of bytes used in the *`result`* buffer.

`error-item`		`output`

`PIC S9(4) COMP`

returns the index of the item being processed when the error occurred. The index starts at 0.

call-error `output`

`PIC S9(4) COMP`

is a variable you declare for the GIVING parameter to store the SQL error code that indicates the results of the call. SQLCAGETINFOLIST procedure error codes are:

Error Code	**Description**
8510	A required parameter is missing.
8511	The program specified an invalid item code.
8512	The program specified an invalid SQLCA structure.
8513	The program specified an SQLCA structure with a version more recent than the version of the SQLCAGETINFOLIST procedure.
8514	Insufficient buffer space is available.
8515	The program specified an error entry index less than zero or greater than the number of errors.
8516	The program specified a *namesmax* parameter less than or equal to zero.
8517	The program specified a *paramsmax* parameter less than or equal to zero.

The item codes you can specify in the *item-list* array are:

Item Code	**Size (Bytes)**	**Description**
1	2	SQLCA version.
2	2	Maximum number of errors or warnings the SQLCA can represent.
3	2	Actual number of errors or warnings.
4	2	Whether there were more errors or warnings than the SQLCA had space to store: 0 = There were no more errors or warnings nonzero = There were more errors or warnings
5	2	Whether there were more parameters than the SQLCA had space to store: 0 = There were no more parameters nonzero = There were more parameters
6	2	Maximum length, in bytes, of the name of the paragraph in which the SQL statement appears.
7	2	Actual length, in bytes, of the name of the paragraph in which the SQL statement appears.
8	(in item code 7)	Program ID of the program in which the SQL statement appears.
9	4	Source code line number of the SQL statement that caused an error.

Item Code	Size (Bytes)	Description
10	2	Syntax error location. If there was no syntax error, SQL returns -1.
11	2	Maximum length, in bytes, of the system procedure that sets the first error or warning.
12	2	Actual length, in bytes, of the system procedure that sets the first error or warning.
13	(in item code 12)	Location of the system procedure that sets the first error or warning.
14	2	Maximum length, in bytes, of the parameter buffer.
15	2	Used bytes in the parameter buffer.
16	(in item code 15)	Parameter buffer.
17	2	Maximum length, in bytes, of the source name buffer.
18	2	Used bytes in the source name buffer.
19	(in item code 18)	Source name buffer.
20	4	Number of processed rows.
21	8	Estimated query cost.
22	2	SQL error or warning number. Error numbers are positive, warning numbers are negative.

23	2	Subsystem ID: First byte is 0. The second byte can be one of these letters:
		S = SQL/MP component:
		SQL compiler
		SQL catalog manager
		SQL executor
		SQLUTIL process
		SQLCI or SQLCI2 process
		F = SQL file system
		D = DP2 disk process
		G = NonStop OS
		R = FastSort program (SORTPROG process)
		L = Load routines
		I = Sequential I/O (SIO) procedures

24	2	Suppress printing this error (0 = False, nonzero = True)
25	2	Offset into the parameters buffer for parameters associated with the call. SQL/MP returns -1 if there are no parameters.
26	2	Number of parameters for this error.

Item Code	**Size (Bytes)**	**Description**
27	2	Sequence in which the error or warning was set.
28	2	Size of the buffer that contains parameters. Each string is delimited by a zero.
29	(in item code 28)	Buffer that contains parameters, delimited by a zero. Each parameter begins on an even word boundary and is preceded by 2 bytes.

Example 5-3 on page 5-23 shows a call to the SQLCAGETINFOLIST procedure that returns these items:

- The name of the procedure containing the SQL/MP statement that produced one or more errors or warnings
- The name length of the procedure name
- The number of errors or warnings that occurred
- The error code of the fifth error returned

Example 5-3. Calling the SQLCAGETINFOLIST procedure (page 1 of 2)

```
WORKING-STORAGE SECTION.
   ...
* Declare a buffer to hold the error information:

 01 ERRORS-AND-WARNINGS.
    02 NAME-LEN   PIC S9(4) COMP.
    02 NUM-ERRS   PIC S9(4) COMP.
    02 NAME       PIC X(32).
    02 ERR-CODE   PIC S9(4) COMP.

* Include the SQLCA declaration:
  EXEC SQL   INCLUDE SQLCA   END-EXEC.

* Declare a variable to hold the size of the buffer, to
* be calculated using INSPECT..TALLYING:
  01 ERR-WARN-SIZE     PIC S9(4) COMP.
* Declare a variable to hold the return code for the call:

  01 CALL-ERROR        PIC S9(4) COMP.

* Declare the item-list table:

  01 ITEM-LIST.
     02 ITEMS          PIC S9(4) COMP OCCURS 4 TIMES.

  01 ERROR-ITEM        PIC S9(4) COMP.
 PROCEDURE DIVISION.

* Initialize the item-list table:

* Code for actual name length:
  MOVE 7 TO ITEMS(1).

* Code for actual number of errors or warnings:
  MOVE 3 TO ITEMS(2).

* Code for procedure name:
  MOVE 8 TO ITEMS(3).

* Code for error number:
  MOVE 22 TO ITEMS(4).

* Calculate the size of the buffer for passing to
* SQLCAGETINFOLIST:
  INSPECT ERRORS-AND-WARNINGS
    TALLYING ERR-WARN-SIZE FOR CHARACTERS.
  ...
```

Example 5-3. Calling the SQLCAGETINFOLIST procedure (page 2 of 2)

```
* Call SQLCAGETINFOLIST.  The third parameter is the number
* of codes you are supplying in item-list.  The seventh
* parameter will cause the program to truncate all names to
* 32 characters.  The ERROR-ITEM parameter receives
* information about the fifth entry in the errors array.

 ENTER TAL "SQLCAGETINFOLIST" USING
            SQLCA,
            ITEM-LIST,
            4,
            ERRORS-AND-WARNINGS,
            ERR-WARN-SIZE,
            OMITTED,
            32,
            OMITTED,
            OMITTED,
            ERROR-ITEM
                       GIVING CALL-ERROR.
```

To avoid hard-coding the maximum length for the procedure name (NAME-LEN OF ERRORS-AND-WARNINGS in this example), perform these steps:

1. Call SQLCAGETINFOLIST and pass item code 6 (the maximum length of procedure name).
2. Call SQLCAGETINFOLIST again and pass a buffer of the appropriate size.

SQLGETCATALOGVERSION

The SQLGETCATALOGVERSION procedure returns the version of a catalog.

```
ENTER TAL "SQLGETCATALOGVERSION" USING
                                [ catalog-name, ]
                                  sql-version
                                  GIVING error-info.
```

catalog-name optional input

PIC X(*length*)

is the fully qualified file name of the catalog for which you are requesting information. The name must be:

- Left justified and padded with spaces on the right
- A maximum of 26 characters

If you omit *catalog-name*, SQLGETCATALOGVERSION uses the default catalog.

`sql-version`   `output`

`PIC S9(4) COMP`

is the version of the catalog. For information about versions of SQL/MP, see the *SQL/MP Version Management Guide*.

`error-info`   `output`

`PIC S9(4) COMP`

indicates the results of the SQLGETCATALOGVERSION call. If the call is successful, *`error-info`* is 0. If an error occurs, *`error-info`* contains the operating system or SQL error number. For a description of SQL errors, see the *SQL/MP Messages Manual*.

Note. Although version 315 SQL/MP software supports the SQLGETCATALOGVERSION procedure, HP might not support this procedure in a future PVU. If you are running version 300 (or later) SQL/MP software, use the GET VERSION OF CATALOG statement to return the version of a catalog. For information about this statement, see the *SQL/MP Reference Manual*.

SQLGETOBJECTVERSION

The SQLGETOBJECTVERSION procedure returns the version of an SQL object.

```
ENTER TAL "SQLGETOBJECTVERSION" USING
                                object-name,
                                sql-version
                                GIVING error-info.
```

`object-name`   `required`   `input`

`PIC X(length)`

is the fully qualified file name of the SQL object for which you are requesting information. The name must be:

- Left justified and padded with spaces on the right
- A maximum of 34 characters

`sql-version`   `output`

`PIC S9(4) COMP`

is the version of the SQL object. For information about versions of SQL/MP, see the *SQL/MP Version Management Guide*.

error-info output

PIC S9(4) COMP

indicates the results of the SQLGETOBJECTVERSION call. If the call is successful, *error-info* is 0. If an error occurs, *error-info* contains the operating system or SQL error number. For a description of SQL errors, see the *SQL/MP Messages Manual*.

Note. Although version 315 SQL/MP software supports the SQLGETOBJECTVERSION procedure, HP might not support this procedure in a future PVU. If you are running version 300 (or later) SQL/MP software, use the GET VERSION statement to return the version of an SQL object. For information about this statement, see the *SQL/MP Reference Manual*.

SQLGETSYSTEMVERSION

The SQLGETSYSTEMVERSION procedure returns the version of the SQL file-system and disk-process components running on a specific node. For a specific node, you can assume that all SQL components are of the same PVU.

If you request the version for a remote node, SQLGETSYSTEMVERSION returns information about the remote disk process. A successful call does not guarantee that SQL/MP is installed on the remote node.

```
ENTER TAL "SQLGETSYSTEMVERSION" USING
                              [ node-number, ]
                                sql-version
                                GIVING error-info.
```

node-number optional input

PIC S9(4) COMP

is the node number for which you are requesting information. If you omit this parameter, SQLGETSYSTEMVERSION returns the version of the local node.

sql-version output

PIC S9(4) COMP

is the SQL/MP software version for the specified system. For information about versions of SQL/MP, see the *SQL/MP Version Management Guide*.

```
error-info                                        output
```

```
PIC S9(4) COMP
```

indicates the results of the call. The procedure returns zero after a successful operation. Otherwise, it returns a nonzero value to indicate an error or warning. For a description of SQL errors, see the *SQL/MP Messages Manual*.

Note. Although version 315 SQL/MP software supports the SQLGETSYSTEMVERSION procedure, HP might not support this procedure in a future PVU. If you are running version 300 (or later) SQL/MP software, use the GET VERSION OF SYSTEM statement to return the version of a system. For information about this statement, see the *SQL/MP Reference Manual*.

SQLSADISPLAY

The SQLSADISPLAY procedure displays the execution statistics of SQL statements in tabular form.

Because the PREPARE statement continually redefines the fields of the SQLSA structure during the execution of dynamic SQL statements, SQLSADISPLAY does not display an SQLSA structure returned by a PREPARE statement.

```
ENTER TAL "SQLSADISPLAY" USING
                           sqlsa,
                         [ sqlca,        ]
                         [ out-file-num, ]
                         [ detail-params ].
```

```
sqlsa                        required        input
```

is the SQLSA to be displayed. The SQLSA is automatically declared in the program when you specify the INCLUDE SQLSA directive.

```
sqlca                        optional        input
```

is the SQLCA that contains the procedure name and line number of the SQL statement that sets the SQLSA to be displayed. If the SQLCA name is not included, the display does not contain the procedure name and process name of the caller. The SQLCA is declared automatically if you specify the INCLUDE SQLCA directive.

```
out-file-num                 optional        input/output
```

```
PIC S9(4) COMP
```

is the output file number. If you omit this value or set it to a negative value, SQLSADISPLAY displays information on your home terminal. SQL/MP ignores this parameter if `detail-params` specifies sequential I/O (SIO).

detail-params optional input

determines whether sequential I/O (SIO) or Enscribe I/O is used for writing to the output file. A COBOL program usually omits *detail-params* and uses Enscribe I/O (the default). The parameter *detail-params* points to a structure with this TAL declaration:

```
STRUCT detail^params;
BEGIN
  sio             STRING;
  out^fcb^1       INT .EXT;
  out^fcb^2       INT .EXT;
END;
```

`sio`

specifies whether sequential I/O is used:

Y	Use SIO; ignore *output-file-number*.
N	Do not use SIO; write to *output-file-number*.

`out^fcb^1`

specifies the first output file control block if SIO is enabled.

`out^fcb^2`

specifies the second output file control block if SIO is enabled. To use this field, assign it a value greater than 0.

Example of the SQLSADISPLAY Display

SQLSADISPLAY displays statistics in this format:

```
SQL statistics @ \system.$vol.subvol.file.#line process cpu,pin

             Records   Records  Disc   Message  Message  Lock
Table Name   Accessed  Used     Reads  Count    Bytes    WE
```

The elements of the SQLSADISPLAY procedure display are:

Element	**Description**
`\`*system.$vol.subvol.file*	The fully qualified file name of the calling program
#*line*	The line number of the calling program
`process` *cpu,pin*	The process ID of the calling program
`Table Name`	The name of each table
`Records Accessed`	The number of records accessed in each table (this includes records examined by the disk process, the file system, and the SQL executor)

Element	**Description**
`Records Used`	The number of records actually used by the statement
`Disc Reads`	The number of disk reads caused by accessing this table
`Message Count`	The number of messages sent to execute operations on this table
`Message Bytes`	The number of message bytes sent to access this table
`Lock WE`	A flag indicating either that lock waits occurred (W) or that lock escalations occurred (E) for the table

Example 5-4 shows the information SQLSADISPLAY displays. To generate this display, a program follows these steps:

1. Generates the SQLSA and SQLCA structures.
2. Executes an SQL DML statement.
3. Calls the SQLSADISPLAY procedure.

Example 5-4. SQLSADISPLAY Display

```
SQL statistics @ \sanfran.$system.accts.prog10.#333.2 process 12,255

               Records   Records   Disc   Message   Message   Lock
Table Name     Accessed  Used      Reads  Count     Bytes     WE

\sanfran.$sqlvol.accts.tab10

               123       22        3      10        3245

\sanfran.$vol001.fy93.employee

               9987231   1         99999  1         100       e

\sanfran.$sqlvol.accts.tab20

               1        1          0      1         100       w
```

Superseded Procedures

The SQLCADISPLAY and SQLCATOBUFFER procedures have been superseded by the SQLCA_DISPLAY2_ and SQLCA_TOBUFFER2_ procedures, respectively. The data type of parameters in the SQLCA_DISPLAY2_ and SQLCA_TOBUFFER2_ make those procedures easier to use in COBOL programs. These procedures are included in this manual for compatibility with earlier PVUs.

SQLCADISPLAY

The SQLCADISPLAY procedure displays the error or warning messages returned to the SQLCA structure. This procedure displays the information to a file or to a terminal.

The error or warning messages can be from these subsystems or system components:

- SQL/MP
- NonStop OS
- File system
- Disk process (DP2)
- FastSort program (SORTPROG process)
- Sequential I/O (SIO) procedures

```
ENTER TAL "SQLCADISPLAY" USING
                         sqlca,
                       [ output-file-number,   ]
                       [ output-record-length, ]
                       [ sql-msg-file-number,  ]
                       [ errors,               ]
                       [ warnings,             ]
                       [ statistics,           ]
                       [ caller-error-loc,     ]
                       [ internal-error-loc,   ]
                       [ prefix,               ]
                       [ prefix-length,        ]
                       [ suffix,               ]
                       [ suffix-length,        ]
                       [ detail-params         ].
```

sqlca required input

is the record name of the SQLCA to be displayed. The SQLCA is declared automatically when you include the INCLUDE SQLCA directive.

output-file-number optional input

`PIC S9(4) COMP`

is the output file number. If you omit this value or set it to a negative value, SQLCADISPLAY displays information at your home terminal. SQL/MP ignores this parameter if *detail-params* specifies sequential I/O (SIO).

`output-record-length` `optional` `input`

`PIC S9(4) COMP`

is the length in bytes of records to be written to the output file. The length must be an integer value from 60 to 600.

The default is 79 bytes.

`sql-msg-file-number` `optional` `input/output`

`PIC S9(4) COMP`

is the file number of the SQL message file (SQLMSG is the default file). If you specify -1 as an input value, the system opens the message file and returns the resulting file number. If you specify a value other than -1, the system uses that value as the file number of the message file.

To improve the performance of a program that makes multiple calls to the SQLCADISPLAY procedure, specify -1 on the first call and then use the returned file number for subsequent calls. By using the file number, the system opens the file only once and uses the file number for subsequent calls. Otherwise, the system opens the file for each call.

The SQLMSG file contains text in English. You can specify a different SQL message file with the =_SQL_MSG_*`system`* DEFINE. For more information, see SQL Message File on page 5-2.

`errors` `optional` `input`

`PIC S9(4) COMP`

controls the display of error messages to the buffer:

Y	Display all errors.
N	Display only the first error.
B	Display all errors but suppress this prefix: `ERROR from` *`subsystem`* `[`*`nn`*`]:`

The default is Y.

`warnings` `optional` `input`

`PIC S9(4) COMP`

controls the display of warning messages to the buffer:

Y	Display all warning messages.
N	Do not display any warning messages.
B	Display all warnings but suppress this prefix: `WARNING from` *`subsystem`* `[`*`nn`*`]`

The default is Y.

`statistics`   `optional`   `input`

`PIC S9(4) COMP`

controls the display of statistics:

Y	Display row and cost statistics if the value returned to the SQLCA in the ROW or COST field is greater than or equal to 0.
N	Do not display statistics.
R	Display row statistics only.
C	Display cost statistics only.

The default is Y.

`caller-error-loc`   `optional`   `input`

`PIC S9(4) COMP`

controls the display of the program name and line number of the SQL statement that received the error:

Y	Display the program name and line number.
N	Suppress the display.

The default is Y.

`internal-error-loc`   `optional`   `input`

`PIC S9(4) COMP`

controls the display of the system-code location where the first error in the SQLCA occurred:

Y	Display the location.
N	Suppress the information.

The default is Y.

`prefix`   `optional`   `input`

`PIC X(`*`length`*`)`

is a string to precede each output line. The default is three asterisks and a space (***).

`prefix-length` `optional` `input`

`PIC S9(4) COMP`

is the length of the *`prefix`* string for each output line. This length must be an integer value from 1 to 15. If you include *`prefix`*, *`prefix-length`* is required.

`suffix` `optional` `input`

`PIC X(`*`length`*`)`

is a string to be appended to each output line. The default is a null string.

`suffix-length` `optional` `input`

`PIC S9(4) COMP`

is the length of the suffix string for each output line. This length must be an integer value from 1 to 15. If you include *`suffix`*, *`suffix-length`* is required.

`detail-params` `input`

`(record)`

determines whether sequential I/O (SIO) or Enscribe I/O is used for writing to the output file. A COBOL program usually omits *`detail-params`* and uses Enscribe I/O (the default). The parameter *`detail-params`* points to a structure with this TAL declaration:

```
STRUCT detail^params;
BEGIN
  sio             STRING;
  out^fcb^1       INT .EXT;
  out^fcb^2       INT .EXT;
END;
```

`sio`

specifies whether sequential I/O is used:

Y	Use SIO; ignore *`output-file-number`*.
N	Do not use SIO; write to *`output-file-number`*.

`out^fcb^1`

specifies the first output file control block if SIO is enabled.

`out^fcb^2`

specifies the second output file control block if SIO is enabled. To use `out^fcb^2`, assign it a value greater than 0.

Example

This SQLCADISPLAY statement uses all default values:

```
ENTER TAL “SQLCADISPLAY” USING SQLCA.
```

SQLCATOBUFFER

The SQLCATOBUFFER procedure writes to a buffer the error or warning messages returned by SQL/MP. The buffer is a record area declared in the Working-Storage or Extended-Storage Section of the program.

The information returned to the buffer can originate from these subsystems or system components:

- SQL/MP
- NonStop OS
- File system
- Disk process (DP2)
- FastSort program (SORTPROG process)
- Sequential I/O (SIO) procedures

This procedure is similar to the SQLCADISPLAY procedure, which writes error information to a file or terminal.

```
ENTER TAL "SQLCATOBUFFER" USING
                            sqlca,
                            output-buffer,
                            output-buffer-length,
                          [ first-record-number,  ]
                          [ output-records,       ]
                          [ more,                 ]
                          [ output-record-length, ]
                          [ sql-msg-file-number,  ]
                          [ errors,               ]
                          [ warnings,             ]
                          [ statistics,           ]
                          [ caller-error-loc,     ]
                          [ internal-error-loc,   ]
                          [ prefix,               ]
                          [ prefix-length,        ]
                          [ suffix,               ]
                          [ suffix-length         ].
```

sqlca required input

is the record name of the SQLCA.

The SQLCA is declared automatically when you specify the INCLUDE SQLCA directive.

output-buffer	required	input/output

`PIC X(`*`length`*`)`

is the record name to which SQLCATOBUFFER writes the error information.

output-buffer-length	required	input

`PIC S9(4)`

is the length of *output-buffer* in bytes. This length must be

- An integer value from *output-record-length* through 600
- A multiple of *output-record-length*

The minimum length recommended is 300 bytes.

first-record-number	optional	input

`PIC S9(4) COMP`

is the ordinal number of the first error record (line) to be moved to the output buffer. The procedure discards any error records with a lower number. The count of lines moved begins with 1.

The default is 1.

To obtain more than one error record, you must increment the value in *first-record-number*.

output-records	optional	output

`PIC S9(4) COMP`

is the number of records (lines) written by SQLCATOBUFFER to *output-buffer*.

more	optional	output

`PIC X`

is a flag that indicates whether all the desired lines fit into the *output-buffer*:

Y	There were additional records; the buffer overflowed.
N	There were no additional records.

output-record-length	optional	input

`PIC S9(4) COMP`

defines the length of records to be written to the *output-buffer*. The length must be an integer value from 60 to 600.

The default is 79 bytes.

The procedure pads each line with spaces and adds the suffix and prefix strings if the ENTER statement specifies them.

sql-msg-file-number	`optional`	`input/output`

`PIC S9(4) COMP`

is the file number of the SQL message file (SQLMSG is the default file). If you specify -1 as the input value, the system opens the message file and returns the resulting file number. If you specify a value other than -1, the system uses that value as the file number of the message file.

To improve the performance of multiple calls to the SQLCA_DISPLAY2_ procedure, specify -1 on the first call and then use the returned file number for subsequent calls. By using the file number, the system opens the file only once and uses the file number for subsequent calls. Otherwise, the system opens the file for each call.

The SQLMSG file contains text in English. You can specify a different SQL message file with the =_SQL_MSG_*system* DEFINE. For more information, see SQL Message File on page 5-2.

errors	`optional`	`input`

`PIC S9(4) COMP`

controls the writing of error messages to the buffer:

Y	Write all errors.
N	Write only the first error.
B	Write all errors but suppress this prefix: `ERROR from` *subsystem* `[`*nn*`]:`

The default is Y.

warnings	`optional`	`input`

`PIC S9(4) COMP`

controls the writing of warning messages to the buffer:

Y	Write all warning messages.
N	Do not write any warning messages.
B	Write all warnings but suppress this prefix: `WARNING from` *subsystem* `[`*nn*`]`

The default is Y.

`statistics` optional input

`PIC S9(4) COMP`

controls the writing of statistics to the buffer:

Y	Write row and cost statistics if the value returned to the SQLCA in the ROW or COST field is greater than or equal to 0.
N	Do not write statistics.
R	Write row statistics only.
C	Write cost statistics only.

The default is Y.

`caller-error-loc` optional input

`PIC S9(4) COMP`

controls the writing of the program name and line number of the SQL statement that received the error:

Y	Write the program name and line number.
N	Suppress the information.

The default is Y.

`internal-error-loc` optional input

`PIC S9(4) COMP`

controls the writing of the system-code location where the first error in the SQLCA occurred:

Y	Write the location.
N	Suppress the information.

The default is Y.

`prefix` optional input

`PIC X(`*`length`*`)`

is a string to precede each output line. The default is three asterisks and a space (***).

`prefix-length` optional input

`PIC S9(4) COMP`

is the length of the *`prefix`* string for each output line. This length must be an integer value from 1 to 15. If you include *`prefix`*, *`prefix-length`* is required.

suffix	optional	input

`PIC X(`*`length`*`)`

is a string to be appended to each output line. The default is a null string.

suffix-length	optional	input

`PIC S9(4) COMP`

is the length of the *suffix* string for each output line. This length must be an integer value from 1 to 15. If you include *suffix*, *suffix-length* is required.

This example uses a 75-character output record and declares a buffer SQLMSG-BUFFER as 375 characters. The ENTER statement specifies the SQLMSG file number with data item SQLMSG-FILENUM set to -1. The statement returns the file number so that subsequent calls retain the number.

```
WORKING-STORAGE SECTION.

01 SQLMSG-BUFFER        PIC X(375).
01 SQLMSG-FILENUM       PIC S9(4) COMP   VALUE -1.

PROCEDURE DIVISION.
  ...
  ENTER TAL "SQLCATOBUFFER" USING
                      SQLCA,
                      SQLMSG-BUFFER,
                      375,
                      OMITTED,
                      OMITTED,
                      OMITTED,
                      75,
                      SQLMSG-FILENUM.
  ...
```

6 Explicit Program Compilation

This section describes the explicit compilation of an HP COBOL program containing embedded SQL statements and directives in the Guardian, OSS, and PC host environments using TNS and TNS/R compilation tools.

Topics include:

Compilation Methods

The HP COBOL compilers translate HP COBOL source programs into machine language that is specific to a particular NonStop system architecture. Therefore, the type of HP COBOL compiler that you use to compile your program determines the NonStop system where you can run the program. For more information, see Table 6-2, COBOL Compilation Mode and Execution Environment, on page 6-12.

You can compile a source program in TNS mode or native mode on a NonStop system. A TNS-compiled program uses TNS process and memory architecture and consists of TNS object code (TNS instructions), whereas a natively compiled program uses native process and memory architecture and consists of native object code (RISC instructions). The steps for compiling an embedded SQL/MP program for each compilation mode are:

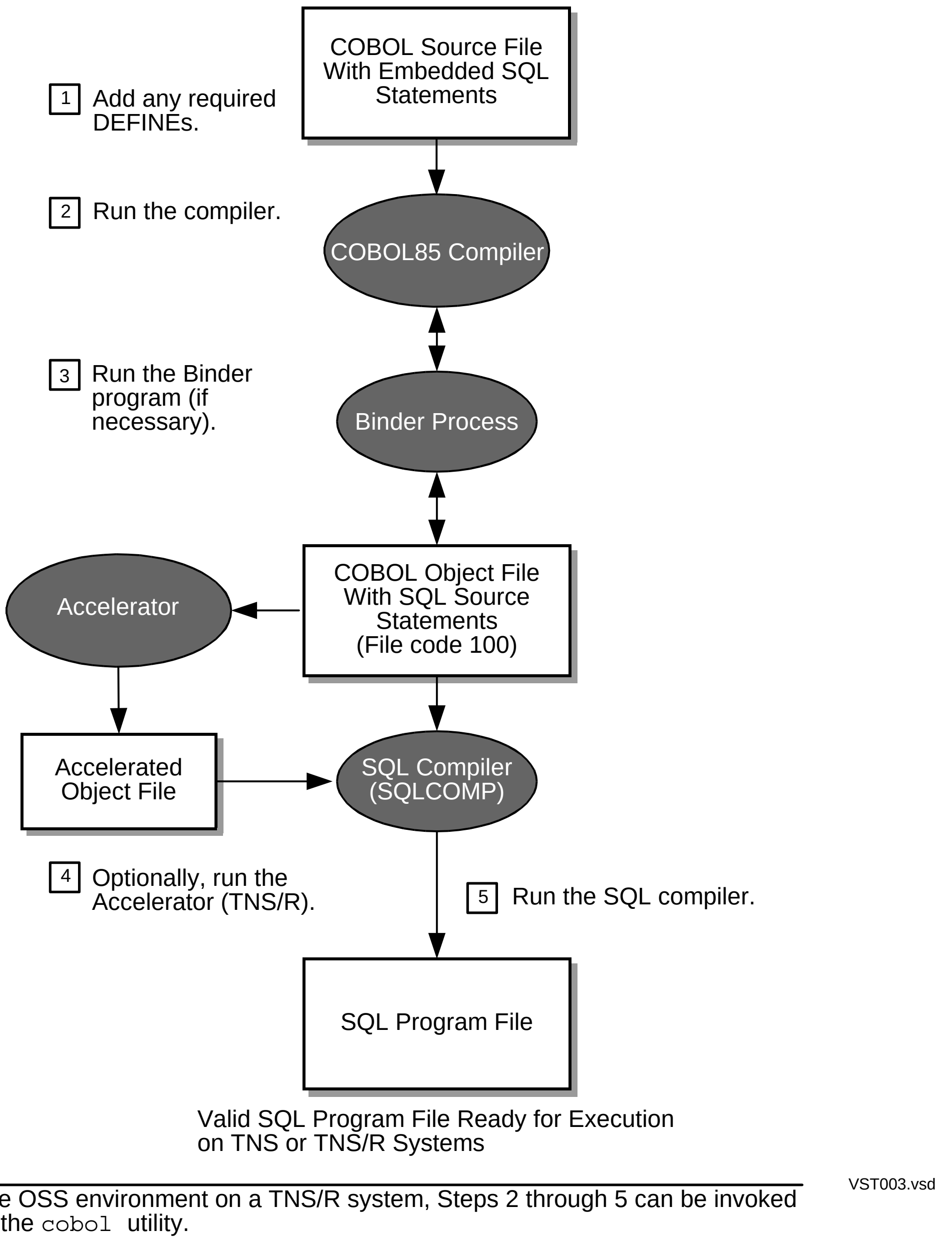
- TNS Mode Compilation on page 6-2
- Native Mode Compilation for TNS/R Systems on page 6-4

TNS Mode Compilation

A TNS-compiled program uses TNS process and memory architecture and consists of TNS object code (TNS instructions). Compiling an HP COBOL program in TNS mode enables you to execute the program on a TNS system. You can also execute TNS programs on TNS/R systems and boost the execution speed on those systems by generating accelerated object code after compilation.

Figure 6-1 shows the steps you follow to explicitly SQL compile a COBOL program in TNS mode.

Figure 6-1. Compiling a COBOL Program in TNS Mode


COBOL Source File With Embedded SQL Statements
1 Add any required DEFINEs.
2 Run the compiler.
COBOL85 Compiler
3 Run the Binder program (if necessary).
Binder Process
Accelerator
COBOL Object File With SQL Source Statements (File code 100)
Accelerated Object File
SQL Compiler (SQLCOMP)
4 Optionally, run the Accelerator (TNS/R).
5 Run the SQL compiler.
SQL Program File
Valid SQL Program File Ready for Execution on TNS or TNS/R Systems
In the OSS environment on a TNS/R system, Steps 2 through 5 can be invoked with the cobol utility.
VST003.vsd

To compile a COBOL program that contains embedded SQL statements and directives for execution on a TNS (or TNS/R) system:

1. Add any required class MAP or class CATALOG DEFINEs.
2. Run the COBOL85 compiler and specify a source file as input.

   Your compilation unit must include an SQLCODE variable declaration (either declared explicitly or implicitly with the INCLUDE SQLCA directive). You must specify the SQL compiler directive in the compilation unit or on the compiler command line.
3. If necessary, use the Binder program to combine the COBOL object file with other object files.
4. Optionally, run the Accelerator on the COBOL object file to optimize it for a TNS/R system.
5. Run the SQL compiler (SQLCOMP) to compile the SQL source statements in the COBOL object file and to produce a valid SQL program file for execution.

The SQL program file that is produced can be executed in either the OSS or Guardian operating environment, depending on how you compile the program:

- In the Guardian environment, execute the SQL program file, either interactively by using the TACL RUN (or RUND) command or programmatically by using the COBOL CREATEPROCESS or CLU_PROCESS_CREATE_ routine.
- In the OSS environment of a TNS/R system, add the directory of the SQL program file to your search path by using the `export` command and then execute the SQL program file by entering its name at the OSS prompt.

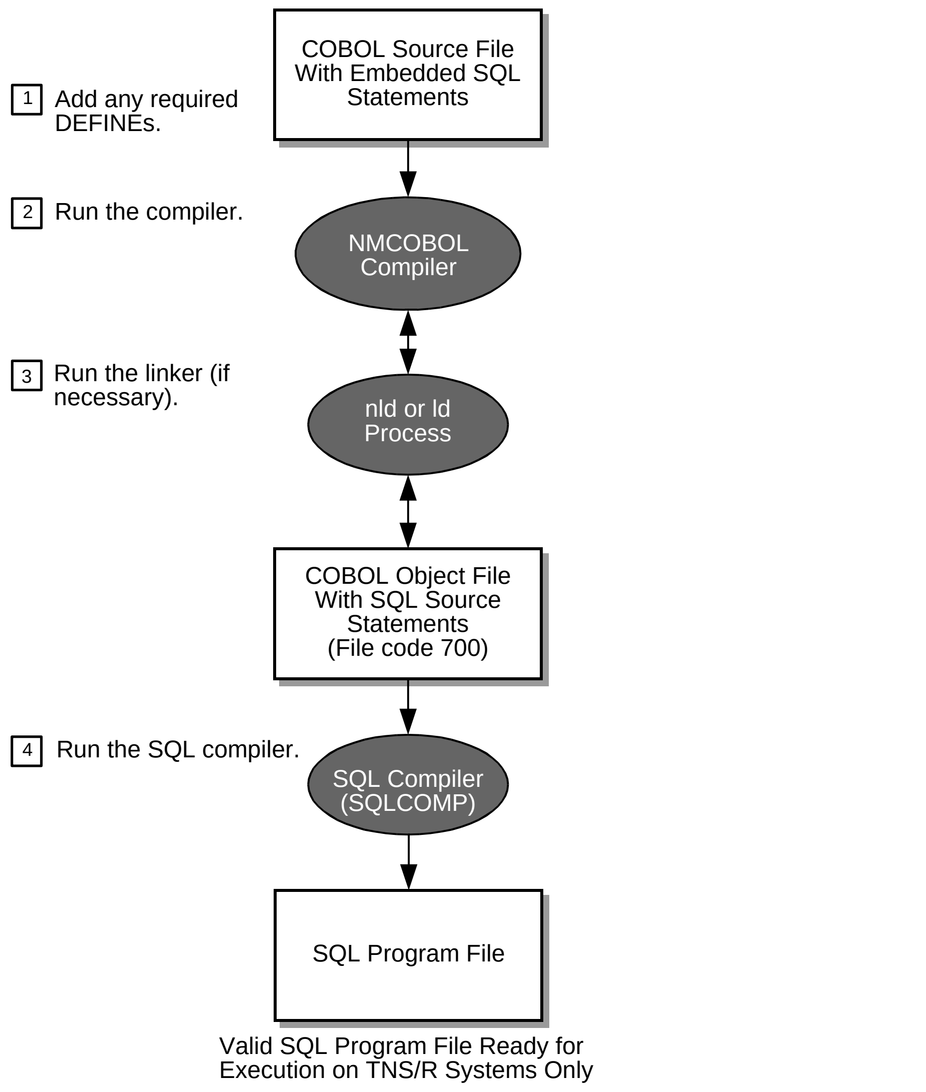
For more information, see Section 7, Program Execution.

Native Mode Compilation for TNS/R Systems

A natively compiled program for a TNS/R system uses native TNS/R process and memory architecture and consists of native object code (RISC instructions). Compiling an HP COBOL program in native TNS/R mode enables you to execute the program on a TNS/R system only.

Figure 6-2 shows the steps you follow to explicitly SQL compile a COBOL program in TNS/R native mode.

Figure 6-2. Compiling a COBOL Program in TNS/R Native Mode


COBOL Source File With Embedded SQL Statements
1 Add any required DEFINEs.
2 Run the compiler.
NMCOBOL Compiler
3 Run the linker (if necessary).
nld or ld Process
COBOL Object File With SQL Source Statements (File code 700)
4 Run the SQL compiler.
SQL Compiler (SQLCOMP)
SQL Program File
Valid SQL Program File Ready for Execution on TNS/R Systems Only
VST003R.vsd

In the OSS environment, Steps 2 through 4 can be invoked with the `nmcobol` utility.

To compile a COBOL program that contains embedded SQL statements and directives for execution on a TNS/R system:

1. Add any required class MAP or class CATALOG DEFINEs.
2. Run the NMCOBOL compiler and specify a source file as input.

   Your compilation unit must include an SQLCODE variable declaration (either declared explicitly or implicitly with the INCLUDE SQLCA directive). You must specify the SQL compiler directive on the compiler command line.
3. If necessary, use the `nld` or `ld` utility to combine the COBOL object file with other object files.
4. Run the SQL compiler (SQLCOMP) to compile the SQL source statements in the COBOL object file and to produce a valid SQL program file for execution.

The SQL program file that is produced can be executed in either the OSS or Guardian operating environment, depending on how you compile the program:

- In the Guardian environment, execute the SQL program file, either interactively by using the TACL RUN (or RUND) command or programmatically by using the COBOL CREATEPROCESS or CLU_PROCESS_CREATE_ routine.
- In the OSS environment, add the directory of the SQL program file to your search path by using the `export` command, and then execute the SQL program file by entering its name at the OSS prompt.

For more information, see Section 7, Program Execution.

Preparing for Compilation

Before compiling an embedded SQL/MP program, verify that the source code is ready for compilation and configure the compilation environment. Follow these guidelines:

- Requirements for Compiling a COBOL Program on page 6-6
- SQL Compiler Directive on page 6-7
- Copying Source Code Into a Compilation Unit on page 6-9
- Setting DEFINEs on page 6-9
- Using PARAM Commands on page 6-11

Requirements for Compiling a COBOL Program

Before compiling the program, verify that the source code contains the required elements for compilation.

Feature or Option	COBOL Compiler Requirements
SQL directive	Required either in the compilation unit before the first Identification Division or on the compiler command line of the COBOL85 compiler (TNS mode) Required on the compiler command line of the NMCOBOL compiler (TNS/R mode) See the SQL Compiler Directive on page 6-7.
SQLMEM directive	Optional in the source code or on the compiler command line of the COBOL85 compiler (TNS mode) See the *COBOL85 for NonStop Systems Manual*.
SQLCODE identifier	Required for each program and nested program. You must declare an SQLCODE identifier either explicitly as a data item or implicitly using the INCLUDE SQLCA directive. See Section 9, Error and Status Reporting.
SQLCODE level-88 items	Optional. You can use level-88 items with an SQLCODE data item by substituting an SQLCODEX data item. See Using the SQLCODEX Data Item on page 9-6.
SQLCA data structure	The compiler: ● Declares an SQLCA structure in a program only if the source file specifies an INCLUDE SQLCA directive. ● Allows an INCLUDE SQLCA directive in the Extended-Storage Section.
SQL statement placement	Allows COBOL statements and embedded SQL statements to be on the same line, except that a COBOL statement must follow the SQL statement terminator, cannot precede an SQL statement, and cannot be a COPY or REPLACE statement.
Library files	Supports the COBOL SOURCE directive.
COPY and REPLACE statements	The compiler implements these restrictions: A COPY or REPLACE statement is not allowed within SQL statements and cannot contain SQL statements. COPY and REPLACE affect SQL statements only between BEGIN DECLARE and END DECLARE directives. ● A COPY statement cannot copy source text that contains SQL statements. ● A REPLACE statement that precedes one or more SQL statements does not affect them, except in a Declare Section.
SQL cursors	Supports local and foreign cursors.
SQL statements in listing	Lists SQL statements in the compiler listing exactly as they appear in the source program.

Feature or Option	COBOL Compiler Requirements
INVOKE and INCLUDE directives	Replaces INVOKE and INCLUDE directives with COBOL data declarations that correspond to the SQL structures being invoked or included.
Inspect debugger	You can use the Inspect debugger (for TNS/R) on a COBOL object file. However, the current source line indicated by the Inspect debugger depends on how you produced the object file. When you use the Inspect debugger on an object file that contains embedded SQL, the current source line indicated by the debugger is the embedded SQL statement itself. See the *Inspect Manual*.
CROSSREF program and CROSSREF directive	Lists any embedded SQL statements that contain referenced and changed variables in TNS HP COBOL programs. The NMCOBOL compiler does not produce a cross-reference listing. If you need one for a native TNS/R program, use the `noft` utility with the XREFPROC flag. See the *nld and noft Manual*.

SQL Compiler Directive

The SQL compiler directive indicates to the HP COBOL compiler that a program contains embedded SQL statements or directives and specifies various options for processing the SQL statements or directives. Use this syntax for the SQL directive:

```
SQL [ sql-option-list | ( sql-option-list )]

sql-option-list is:
  sql-option[, sql-option]...

sql-option is:

  PAGES num-pages
  | SQLMAP
  | WHENEVERLIST
  | { RELEASE1 | RELEASE2 }
```

`PAGES` *`num-pages`*

specifies the number of 2048-byte pages of memory the compiler should allocate to the SQL compiler interface (SCI) to process SQL statements and directives. The default (and minimum) value for *`num-pages`* is 560 (on TNS/R systems). The maximum value is 1000.

`SQLMAP`

directs the compiler to include an SQL map in the listing file. An SQL map contains this information:

- Each run-time data unit (RTDU), which is a region of the object file that contains both SQL source statements and object code

- Section location table (SLT) index number, which maps a single SQL statement to a table in the RTDU
- Source file name and number
- Source file line number

The SQL map is sorted first by RTDU name and then by SLT index number. You can use this map to correlate MEASURE output with the SQL statements.

The SQLMAP option also directs the compiler to include the HOSV version in the compiler listing. For example:

```
Host Object SQL Version = 315
```

The default is not to include the SQL map in the listing.

`WHENEVERLIST`

directs the compiler to write active WHENEVER options to the listing file after each embedded SQL statement is processed.

The default is not to write the WHENEVER options.

`RELEASE1 or RELEASE2`

specifies the version of the SQL/MP features in the program (including the SQL data structures) and the version of SQL/MP software on which the program file can run.

`RELEASE1`

specifies version 1 features. A program that uses the RELEASE1 option is compatible with SQL/MP version 1, 2, or 300 (or later) software. This option applies to the COBOL85 compiler only and does not apply to the NMCOBOL compiler.

`RELEASE2`

specifies version 2 features. A program that uses the RELEASE2 option is compatible with SQL/MP version 2 or 300 (or later) software, but not with version 1 software. RELEASE2 is the default.

Note. Although the compiler allows the RELEASE1 and RELEASE2 options, these options might not be supported in a future RVU. If you are using a version 300 (or later) compiler to generate version 1 or version 2 data structures, use the INCLUDE STRUCTURES directive with the VERSION 1 or VERSION 2 option rather than the RELEASE1 or RELEASE2 option. For more information, see Using the INCLUDE STRUCTURES Directive on page 9-1.

Copying Source Code Into a Compilation Unit

To copy the COBOL source code from a separate file into a compilation unit, use one of these options:

- COBOL SOURCE directive
- COBOL COPY statement

You cannot use SQL statements in a file you are copying with the COPY statement. For information about the COPY statement or SOURCE directive, see the *COBOL85 for NonStop Systems Manual*.

Setting DEFINEs

You can use DEFINE names in an SQL program to specify the names of SQL catalogs and objects (tables, views, indexes, partitions, and collations). You must set all DEFINE names used in SQL statements before SQL load time (the time when an SQL program executes its first statement) unless your program uses execution-time name resolution.

- Using DEFINEs in the Guardian Environment
- Using DEFINEs in the OSS Environment on page 6-10

Using DEFINEs in the Guardian Environment

Use a class CATALOG DEFINE for a catalog and a class MAP DEFINE for an object:

- To use DEFINEs, the DEFMODE attribute must be ON for your TACL process. To determine the DEFMODE setting, enter the SHOW DEFMODE command at the TACL prompt:

```
10> SHOW DEFMODE
        Defmode OFF
```

  If DEMODE is OFF, enter a SET DEFMODE ON command:

```
11> SET DEFMODE ON
```

- Before you run the HP COBOL compiler, add the DEFINEs for the names of SQL objects you use in INVOKE directives.

```
12> ADD DEFINE =employee, CLASS MAP, FILE persnl.employee
13> ADD DEFINE =emplist,  CLASS MAP, FILE persnl.emplist
...
```

- Before you run the SQL compiler (SQLCOMP), add the DEFINEs for the names of tables, view, indexes, or collations you use in SQL statements.

```
20> ADD DEFINE =dept,     CLASS MAP, FILE persnl.dept
21> ADD DEFINE =xempname, CLASS MAP, FILE persnl.xempname
```

```
22> ADD DEFINE =collate1, CLASS MAP, FILE collate1
...
```

Note. For information on adding DEFINEs in the OSS environment, see Using DEFINEs in the OSS Environment on page 6-10.

If you specify a DEFINE name in an SQL statement that is not in your current set of DEFINEs, the SQL compiler issues a warning message and leaves the statement uncompiled in the object file. When you run your program, the SQL executor automatically tries to recompile the SQL statement. If the DEFINE is still not available at run time, the SQL compiler issues an error message.

- When you run the SQL compiler, you can specify a CLASS SPOOL DEFINE for the OUT file and a class CATALOG DEFINE for the catalog option. If you use these DEFINEs, add them before you enter the SQLCOMP command:

```
30> ADD DEFINE =persnl, CLASS CATALOG, SUBVOL persnl
31> ADD DEFINE =sqlist, CLASS SPOOL, LOC $S.#sqlist
32> SQLCOMP /IN sqlcbprg,OUT =sqlist,NOWAIT/ CATALOG =persnl
```

- To use the DEFINEs stored in the program file when you explicitly recompile a program, specify the STOREDDEFINES option of the SQLCOMP command. For a description of the STOREDDEFINES option, see SQL Compiler Options on page 6-28.

Using DEFINEs in the OSS Environment

Use these OSS utilities to create and manipulate class MAP and class CATALOG TACL DEFINEs in the OSS environment:

`add_define`	Creates a new class MAP, CATALOG, SPOOL, SORT, SUBSORT, SEARCH, or TAPE DEFINE
`del_define`	Deletes one or more DEFINEs
`info_define`	Displays the attributes and values of existing DEFINEs
`set_define`	Sets the values for one or more DEFINE attributes in the current working attribute set
`show_define`	Displays the values for one or more DEFINE attributes in the current working attribute set

Although you run these utilities in the OSS environment, each utility uses Guardian conventions for its DEFINE attribute and the associated values. For a detailed description, including the syntax of these utilities, see the *Open System Services Shell and Utilities Reference Manual*.

Considerations for using TACL DEFINEs in the OSS environment are:

- The `add_define` utility implicitly sets the DEFMODE attribute to ON before it creates the new DEFINE.
- Before you run the compiler using the `cobol` or `nmcobol` utility, add these DEFINEs:

    - Class MAP DEFINEs specified in INVOKE directives
    - Class MAP or class CATALOG DEFINEs specified in SQL statements

- If you specify a class CATALOG DEFINE for the SQLCOMP CATALOG option when you run the SQL compiler using the `cobol` or `nmcobol` utility, add the DEFINE before you issue the `cobol` or `nmcobol` command.
- You must precede a backslash (\) in a system name or a dollar sign ($) in a catalog or subvolume name with the OSS shell escape character (\). For example, these commands create a class MAP DEFINE and a class CATALOG DEFINE:

```
add_define =emptab class=map file=\\ny.\$disk2.fy94.empfile
add_define =sqlcat class=catalog subvol=\$sql.sqlcat
```

- Use seven characters or fewer for system names or the names of volumes where OSS objects reside.
- To alter an existing DEFINE, use the `add_define` utility and specify all DEFINE attributes and their new values. In this situation, the `add_define` utility essentially adds a new DEFINE with the same name in place of the old DEFINE.

Using PARAM Commands

If you choose to use a PARAM command, you must enter it before you enter the command to run the compiler. The HP COBOL compilers accept these command interpreter PARAM commands:

PARAM Command	Accepted by...
PARAM BINSERV	COBOL85 compiler
PARAM SAMECPU	COBOL85 compiler
PARAM SWAPVOL	COBOL85 and NMCOBOL
PARAM SYMBOL-BLOCKS	COBOL85 and NMCOBOL compilers
PARAM SYMSERV	COBOL85 compiler

A PARAM command does not apply to automatic SQL recompilation or dynamic SQL compilation. For more information about using PARAM commands during compilation, see the *COBOL85 for NonStop Systems Manual*. For the syntax of the PARAM command, see the *TACL Reference Manual*.

Running the HP COBOL Compilers

The type of HP COBOL compiler that you can use to compile an embedded SQL/MP program depends on your operating environment and platform. Table 6-1 lists the HP COBOL compilers, their compilation mode, and the environment and server on which you can run the compilers.

Table 6-1. HP COBOL Compilers

Compiler	Compilation Mode	Operating Environment of the Compiler	NonStop Server of the Compiler
COBOL85	TNS	Guardian	D-series G-series
`cobol`	TNS	OSS	D-series G-series
NMCOBOL	TNS/R native	Guardian	D-series G-series
`nmcobol`	TNS/R native	OSS	D-series G-series
Native COBOL cross compiler for TNS/R	TNS/R native	PC	D-series host* G-series host

* The HP Enterprise Toolkit—NonStop Edition (ETK) and PC command line are not supported on D-series servers. Instead, use the native COBOL cross compiler of the HP Tandem Development Suite (TDS). For more information, see the *COBOL85 for NonStop Systems Manual*.

The compilation mode that you use, depending on your choice of an HP COBOL compiler, determines where you can run the program, as Table 6-2 shows.

Table 6-2. COBOL Compilation Mode and Execution Environment

Compilation Mode	NonStop System Where you can run the Program
TNS	TNS system (C-series servers) TNS/R system (D-series and G-series servers)
TNS/R native	TNS/R system (D-series and G-series servers)

Before compiling an embedded SQL/MP program, verify that the source code is ready for compilation and configure the compilation environment. For more information, see Preparing for Compilation on page 6-5.

To use an HP COBOL compiler to compile an embedded SQL/MP program, see:

- Running HP COBOL Compilers in the Guardian Environment on page 6-13
- Running HP COBOL Compilers in the OSS Environment on page 6-16
- Running the Native COBOL Cross Compilers in a PC Host Environment on page 6-21

Running HP COBOL Compilers in the Guardian Environment

To run an HP COBOL compiler in the Guardian environment, see:

- Running the COBOL85 Compiler in the Guardian Environment
- Running the NMCOBOL Compiler in the Guardian Environment on page 6-14

Running the COBOL85 Compiler in the Guardian Environment

To run the COBOL85 compiler in the Guardian environment, enter the COBOL85 command at the TACL prompt or from a TACL OBEY command file using this syntax:

```
COBOL85 /IN source-file [, OUT [list-file]][, run-option].../
         [target-file ]
         [, copy-library ]
         [; compiler-directive ] ...
```

For more information about the syntax, see the *COBOL85 for NonStop Systems Manual*.

Note. To terminate a compilation in the Guardian environment, use the Break key to return to the command interpreter and stop the process using its process identification number (PIN).

For example, this command invokes the COBOL85 compiler and specifies a source file, MYSRC, which contains embedded SQL statements and directives:

```
COBOL85 /IN MYSRC/
```

The source file can be a disk file, terminal, magnetic tape unit, or process.

By default, the compiler generates an object file, RUNUNIT, qualified by the default system, volume, and subvolume names. To name the object file, specify a target file on the command line. For example, this command generates an object file, MYPROG:

```
COBOL85 /IN MYSRC/ MYPROG
```

To compile an embedded SQL program, you must specify the SQL directive, which tells the compiler to expect SQL statements in the compilation unit. If you do not specify the SQL directive before the first Identification Division in the source program, you must specify it on the compiler command line. For example, this SQL directive tells the compiler to expect embedded SQL statements, to accept only version 1 features of SQL/MP, and to include an SQL map in the listing file, $VOL1.SUBVOL.LST:

```
COBOL85 /IN MYSRC, OUT $VOL1.SUBVOL.LST/ MYPROG; SQL (RELEASE1,
SQLMAP)
```

For the syntax of the SQL directive, see the SQL Compiler Directive on page 6-7.

When compiling an embedded SQL program using the COBOL85 compiler, you can optionally use the SQLMEM directive to cause the compiler to declare SQL data structures in either the Working-Storage Section or the Extended-Storage Section of the program. For example, this command directs the compiler to declare SQL data structures in the Extended-Storage Section:

```
COBOL85 /IN MYSRC/ MYPROG; SQL; SQLMEM EXT
```

For more information about the compiler directives, see the *COBOL85 for NonStop Systems Manual*.

When you run the COBOL85 compiler, it automatically invokes the BINSERV process of the Binder program, which validates and resolves references to other programs or routines and produces a single object file. To invoke the Binder program separately, use BIND. For more information, see The Binder Program on page 6-22.

Note. Run the Binder program before SQL compiling the program.

After compiling and binding the object file, you can optimize the TNS object file for execution on a TNS/R system by invoking the Accelerator. For more information, see Acceleration of TNS HP COBOL Programs on page 6-23.

△ **Caution.** Because the Accelerator invalidates SQL program files, run the Accelerator before you explicitly SQL compile the program to avoid having to recompile.

Finally, you must run the SQL compiler to generate SQL object code in the program file. For more information, see Running the SQL Compiler in the Guardian Environment on page 6-27.

Running the NMCOBOL Compiler in the Guardian Environment

To run the NMCOBOL compiler in the Guardian environment, enter the NMCOBOL command at the TACL prompt or from a TACL OBEY command file using this syntax:

```
NMCOBOL /IN source-file [, OUT [list-file]][, run-option].../
        [ target-file ]
        [, copy-library ]
        [; compiler-directive ] ...
```

For more information about the syntax, see the *COBOL85 for NonStop Systems Manual*.

Note. To terminate a compilation in the Guardian environment, use the Break key to return to the Command Interpreter and stop the process using its process identification number (PIN).

For example, this command invokes the NMCOBOL compiler and specifies a source file, MYSRC, which contains embedded SQL statements and directives:

```
NMCOBOL /IN MYSRC/; SQL
```

The source file must be an EDIT file.

By default, the compiler generates an object file, RUNUNIT, qualified by the default system, volume, and subvolume names. To name the object file, specify a target file on the command line. For example, this command generates an object file, MYPROG:

```
NMCOBOL /IN MYSRC/ MYPROG; SQL
```

For natively compiled programs, the SQL directive is not accepted in the source code. You must specify the SQL directive on the compiler command line as the previous example shows.

The NMCOBOL compiler does not automatically invoke the linker. You must specify the RUNNABLE directive either in the source code or on the compiler command line for the NMCOBOL compiler to call the linker to produce an executable object file (loadfile):

```
NMCOBOL /IN MYSRC/ MYPROG; SQL; RUNNABLE
```

If you specify the RUNNABLE directive but not the CALL-SHARED or SHARED directive, the NMCOBOL compiler automatically links the program by using the COBOLFE process and the `nld` utility to produce a non-PIC loadfile.

If you specify the RUNNABLE and CALL-SHARED directives, the NMCOBOL compiler automatically links the program by using the `ld` utility to produce a PIC loadfile:

```
NMCOBOL /IN MYSRC/ MYPROG; SQL; RUNNABLE; CALL-SHARED
```

Note. Embedded SQL/MP programs are disallowed in user libraries, shared run-time libraries (SRLs), and dynamic-link libraries (DLLs). When compiling embedded SQL/MP programs, do not use the SHARED directive, which produces a DLL.

If you do not specify the RUNNABLE directive when compiling the source program, you must invoke the linker directly to link the object files. For more information, see The nld or ld Utility on page 6-23.

Note. Run the linker before SQL compiling the program.

Finally, you must run the SQL compiler to generate SQL object code in the program file. For more information, see Running the SQL Compiler in the Guardian Environment on page 6-27.

Running HP COBOL Compilers in the OSS Environment

To run an HP COBOL compiler in the OSS environment, see:

- Changing Default Path Names and Disk Volume in the OSS Environment
- Running the cobol Utility in the OSS Environment on page 6-17
- Running the nmcobol Utility in the OSS Environment on page 6-19

Changing Default Path Names and Disk Volume in the OSS Environment

Table 6-3 lists the default path names of the programs that the `cobol` or `nmcobol` commands invoke and the default disk volume on which these programs create temporary files.

Table 6-3. Environment Variables in the OSS Environment

Variable	Effect	Default
`COBOL85`	Determines the path name of the COBOL85 compiler that the `cobol` utility invokes	`/G/system/system/cobol85`
`NMCOBOL`	Determines the path name of the NMCOBOL compiler that the `nmcobol` utility invokes	`/G/system/system/cobolfe`
`BIND`	Determines the path name of the Binder that the `cobol` utility invokes	`/G/system/system/bind`
`NLD`	Determines the path name of the `nld` utility that the `nmcobol` utility invokes	`/usr/bin/nld`
`LD`	Determines the path name of the `ld` utility that the `nmcobol` utility invokes	`/usr/bin/ld`
`AXCEL`	Determines the path name of the Accelerator that the `cobol` utility invokes	`/G/system/system/axcel`
`SQLCOMP`	Determines the path name of the SQL compiler that the `cobol` or `nmcobol` utility invokes	`/G/system/system/sqlcomp`
`SWAPVOL`	Determines the disk volume on which the COBOL85 compiler, Binder, Accelerator, and SQL compiler create temporary files	Same as in the Guardian environment—see the *COBOL85 for NonStop Systems Manual*.
`TMPDIR`	Determines the path name that overrides the default directory for temporary files created by the `nmcobol` utility and the components that it invokes	`/tmp`

To change one or more of the defaults before executing the `cobol` or `nmcobol` command, use the `export` command. The effect of the `export` command lasts until you explicitly change the value of the `export` command.

To execute a `cobol` or `nmcobol` command with a specified set of environment variables, use the OSS `env` function with the environment variables listed in Table 6-3. The effect of the `env` function applies only to the `cobol` or `nmcobol` utility command with which you use it.

For the syntax of the `export` command and the `env` function, see the *Open System Services Shell and Utilities Reference Manual*.

Running the cobol Utility in the OSS Environment

The OSS utility `cobol` generates COBOL programs that run in the OSS environment of TNS or TNS/R systems. The `cobol` utility invokes the COBOL85, optionally followed by the Binder, Accelerator, and SQL compiler. The flags and the types of files in the operands determine which processes operate on the files in the operands.

Text file inputs to the compiler can be OSS ASCII text files (code 180) or Guardian EDIT files (code 101). Embedded SQL/MP source code can be in one of these OSS file types (identified by the file suffix):

`.cbl`
`.cob`
	COBOL source program to be compiled and optionally bound

`.o`	Object file produced by a previous COBOL compilation to be directly passed to the Binder
`.a`	Archive, typically produced by the `ar` utility of nonexecutable linkfiles to be directly passed to the Binder

Note. Embedded SQL/MP programs are disallowed in user libraries, shared run-time libraries (SRLs), and dynamic-link libraries (DLLs).

The command syntax for running the COBOL85 compiler in the OSS environment follows. The `cobol` utility is case-sensitive. Bracketed items are optional. Insert spaces between flags and their parameters, but do not insert spaces on either side of equal signs.

```
cobol  [ flag  [ flag  ]... ] operand ...
```

For detailed information on the `cobol` utility, see the *Open System Services Shell and Utilities Reference Manual*.

Note. To terminate a compilation in the OSS environment, press the Control and c keys (Ctrl-c) simultaneously.

These examples show how to compile an embedded SQL/MP program by using the `cobol` utility:

- To compile, bind, and SQL compile an embedded SQL/MP program, use this type of command:

  ```
  cobol -o /usr/mydir/myprog -L /nonnative/usr/lib
  -Wcobol="SQL" -Wsql="catalog \$vol.subvol" mysrc.cbl
  ```

  - The `-Wcobol` flag directs the `cobol` utility to pass a string of compiler directives to the COBOL85 compiler. In this case, the SQL directive tells the COBOL85 compiler to expect embedded SQL in the source file. The SQL directive is required either in the source code or on the compiler command line. For more information, see the SQL Compiler Directive on page 6-7.
  - The `-Wsql` flag directs the `cobol` utility to invoke the SQL/MP compiler (SQLCOMP), passing the specified arguments to the SQL/MP compiler. In this case, a catalog, $vol.subvol, is passed to the SQL/MP compiler.

    Note. Do not use the `-Wsql` and `-s` flags in the same invocation of the `cobol` utility. The `-s` option strips symbols information from the object file. The SQL compiler requires the symbols region to be present for SQL compilation to succeed.

  - The `-o` flag directs the `cobol` utility to use the specified path name instead of default `a.out` for the executable file produced.
  - The `-L` flag specifies the `/nonnative/usr/lib` directory, which contains `libc.a` and other `.a` files that Binder requires to function properly.

- To accelerate an object file for TNS/R systems, after compiling and binding and before SQL compiling the program file, use this type of command:

  ```
  cobol -o /usr/mydir/myprog -L /nonnative/usr/lib
  -Waxcel="StmtDebug" -Wsql mysrc.cob
  ```

  The `-Waxcel` flag directs the `cobol` utility to invoke the Accelerator, passing the specified arguments to the Accelerator. For more information, see the *Accelerator Manual*.

- By default, the `cobol` utility automatically invokes the BINSERV process of the Binder to bind object files into an executable object file. To suppress the invocation of the Binder, use the `-c` or `-Wnobind` flag on the `cobol` command line:

  - In this example, the `-c` flag directs the `cobol` utility to compile the specified source file, `mysrc.cob`, but not to bind it or remove the object file, `mysrc.o`, that is created in the current directory:

    ```
    cobol -c mysrc.cob
    ```

  - This command invokes the SQL/MP compiler to SQL compile the program file, `myprog`, without invoking the Binder:

    ```
    cobol -Wnobind -Wsql myprog
    ```

- To invoke the Binder program to bind object files into a program file and then invoke the SQL/MP compiler to SQL compile the program file, use this command:

  ```
  cobol -o myprog -L /nonnative/usr/lib
  -Wbind="set heap_max 64" -Wsql x.o y.o z.o
  ```

  The `-Wbind` flag directs the `cobol` utility to pass arguments to the Binder. In the previous example, the Binder sets the maximum heap size to 64 pages. For more information, see the *Binder Manual*.

  Note. Run the Binder program before SQL compiling the program.

Running the nmcobol Utility in the OSS Environment

The OSS utility `nmcobol` generates native COBOL programs that run in the OSS environment of TNS/R systems. The `nmcobol` utility invokes the native HP COBOL compiler, NMCOBOL, optionally followed by the linker and SQL compiler. The flags and the types of files in the operands determine which processes operate on the files in the operands.

Text file inputs to the compiler can be OSS ASCII text files (code 180) or Guardian EDIT files (code 101). Embedded SQL/MP source code can be in one of these OSS file types (identified by the file suffix):

`.cbl` `.cob`	COBOL source program to be compiled and optionally linked
`.o`	Object file produced by a previous COBOL compilation to be passed directly to the linker
`.a`	Archive, typically produced by the `ar` utility, of nonexecutable linkfiles to be passed directly to the linker

Note. Embedded SQL/MP programs are disallowed in user libraries, shared run-time libraries (SRLs), and dynamic-link libraries (DLLs). When compiling embedded SQL/MP programs, do not use the `-Wshared` flag, which produces a DLL.

The command syntax for running the NMCOBOL compiler in the OSS environment follows. The `nmcobol` utility is case-sensitive. Bracketed items are optional. Put spaces between flags and their parameters, but do not put spaces on either side of equal signs.

```
nmcobol  [ flag  [ flag  ]... ] operand ...
```

For detailed information on the `nmcobol` utility, see the *Open System Services Shell and Utilities Reference Manual*.

Note. To terminate a compilation in the OSS environment, press the Control and c keys (Ctrl-c) simultaneously.

These examples show how to compile an embedded SQL/MP program by using the `nmcobol` utility:

- To compile, link, and SQL compile an embedded SQL/MP program, use this command:

  ```
  nmcobol -o /usr/mydir/myprog -Wsql="WHENEVERLIST"
  -Wsqlcomp="catalog \$vol.subvol" mysrc.cbl
  ```

  - The `-Wsql` flag tells the NMCOBOL compiler to expect embedded SQL in the source file and passes optional SQL directive options, such as WHENEVERLIST, to the NMCOBOL compiler. For natively compiled programs, the SQL directive is not accepted in the source code but is on the compiler command line during compilation. For more information, see the SQL Compiler Directive on page 6-7. The `-Wsql` flag also invokes the `-Wsqlcomp` flag if you do not specify it on the command line.

    Note. Do not use the `-Wsql` and `-s` flags in the same invocation of the `nmcobol` utility. The `-s` option strips symbols information from the object file. The SQL compiler requires the symbols region to be present for SQL compilation to succeed.

  - The `-Wsqlcomp` flag directs the `nmcobol` utility to invoke the SQL/MP compiler (SQLCOMP) after the linking step, passing the specified arguments to the SQL/MP compiler. In this case, a catalog, $vol.subvol, is passed to the SQL/MP compiler.

  - The `-o` flag directs the `nmcobol` utility to use the specified path name instead of default `a.out` for the executable file produced.

- By default, the `nmcobol` utility automatically invokes the `nld` utility to link object files into a non-PIC object file (loadfile). To create a PIC program loadfile, use the `-Wcall_shared` flag to invoke the `ld` utility:

  ```
  nmcobol -o /usr/mydir/myprog -Wcall_shared -Wsql mysrc.cbl
  ```

  You can optionally pass arguments to the `ld` utility using the `-Wld` or `-Wld_obey` flag. For more information, see the *ld and rld Reference Manual*.

- To suppress the invocation of the linker, use the `-c` or `-Wnolink` flag on the `nmcobol` command line:

  - In this example, the `-c` flag directs the `nmcobol` utility to compile the specified source file, `mysrc.cob`, but not to link it or remove the object file, `mysrc.o`, that is created in the current directory:

    ```
    nmcobol -c mysrc.cob
    ```

  - This command invokes the SQL/MP compiler to SQL compile the program file, `myprog`, without invoking the linker:

    ```
    nmcobol -Wnolink -Wsql myprog
    ```

- To invoke the `nld` utility to link object files into a non-PIC program file, and then invoke the SQL/MP compiler to SQL compile the program file, use this command:

  ```
  nmcobol -o myprog -Wnld="set heap_max 64" -Wsql x.o y.o z.o
  ```

The `-Wnld` or `-Wnld_obey` flag directs the `nmcobol` utility to pass arguments to the `nld` utility. In the previous example, the `nld` utility sets the maximum heap size to 64 pages. For more information, see the *nld and noft Manual*.

Note. Run the linker before SQL compiling the program.

-Wsqlconnect

This option instructs the compiler about which security mode must be used while communicating with the NSK host. This option works with compilers supported on windows operating system. For example: ecobol.

The syntax is:

```
-Wsqlconnect = mode
```

Where `mode` is:

`legacy`	Directs the compiler to connect using the legacy (unencrypted) mode.
`secure_quiet`	Directs the compiler to connect using the secure (encrypted) mode. If a secure connection cannot be established, the compiler uses the legacy mode. This option does not generate any diagnostics.
`secure_warn`	Directs the compiler to connect using the secure (encrypted) mode. If a secure connection cannot be established, the compiler uses the legacy mode. A warning message is generated when this option is used. This is the default option.
`secure_err`	Directs the compiler to connect using the secure (encrypted) mode. If a secure connection cannot be established, an error occurs, and the compilation terminates.

Usage Considerations

This option requires both the `-Wsqlhost` and `-Wsqluser` options to be specified. If an invalid value is specified, an error is returned.

Using the secure connection mode can increase the compilation time of modules with embedded SQL/MP, by up to a factor of two. This is due to the cost of performing encryption and decryption by using Secure Shell(SSH) or Secure Sockets Layer(SSL), or both. (SQL/MP compilations use both SSL and SSH).

For more information about NSK security, see the *Security Management Guide*.

HP_NSK_CONNECT_MODE

This environment variable is introduced in H06.25/J06.07 RVU and can be set to any of the following values:

- `legacy`
- `secure_quiet`
- `secure_warn`
- `secure_err`

If the environment variable is set to any of the previous values, these values are used by the compiler to set the connection mode. If the environment variable is set to any other value, the compiler returns an error.

If both the –`Wsqlconnect` option is specified and the environment variable is set, the value specified in the option overrides the value set in the environment variable.

Running the Native COBOL Cross Compilers in a PC Host Environment

By using these tools, you can use native COBOL cross compilers to build embedded SQL/MP applications on a PC:

- HP Enterprise Toolkit—NonStop Edition (ETK), which provides a graphical user interface (GUI)
- Command-line interface in Microsoft Windows
- HP Tandem Development Suite (TDS) on D-series servers

After building applications on a PC, you transfer them to the Guardian or OSS environment of a NonStop server for use in production.

A native COBOL cross compiler enables you to build embedded SQL/MP applications for execution on TNS/R systems. The native COBOL cross compiler for TNS/R uses the `nmcobol` utility as its driver.

To perform SQL/MP operations on a PC, you must be connected to a NonStop host. For information about using the native COBOL cross compilers, see:

- ETK online help
- *Using the Command-Line Cross Compilers on Windows* on the native COBOL cross compiler CD or in the ETK online help under References
- *COBOL85 for NonStop Systems Manual*

Binding and Linking

Binding TNS object files or linking native object files involves validating and resolving references to other programs or routines and collecting and modifying code and data blocks from one or more object files to produce a single object file.

The Binder or linker is a tool that you can use to read, link, modify, and build executable object files. Follow these guidelines when you bind or link SQL program files:

- Handle SQL program files like other object files.
- Bind or link object files after they are compiled by the HP COBOL compiler.

  You can bind or link object files after running the SQL compiler. However, the binding or linking operation invalidates the resulting target file, and you must then explicitly recompile the program file to validate it.
- SQL compile only the final bound or linked object. You are not required to separately SQL compile each object of a multiple-module program.
- Give a COBOL program a unique program name if you plan to bind or link it with other programs.

The type of binding or linking process that occurs during compilation or that you can use after compilation depends on the compilation mode that you use. For more information, see:

- The Binder Program on page 6-22
- The nld or ld Utility on page 6-23

The Binder Program

Binding applies only to object files created in TNS mode by the COBOL85 compiler (or by the `cobol` utility in the OSS environment of a TNS/R system). By default, the COBOL85 compiler invokes the Binder program to bind the TNS object files. However, in some cases, you might want to bind object files after compilation. For example, you might want to replace one version of a program (in a object file that contains blocks from several programs) with a new version of the program. The BIND command allows you to invoke the Binder program interactively in the Guardian environment.

To run the Binder program interactively, enter BIND at the TACL prompt. The Binder program displays its banner and prompt, an at sign (@). In this example, the Binder commands combine the COBOBJ1 and COBOBJ2 files into an executable object file, PROGFILE. The SELECT LIST * OFF command improves system performance by turning off all listings.

```
:BIND

@ADD * FROM cobobj1
@ADD * FROM cobobj2
```

```
@SELECT LIST * OFF
@BUILD progfile
@EXIT
```

△ **Caution.** The Binder STRIP command without the SYMBOLS or AXCEL option removes the Binder table from an object file. Without the Binder table, the SQL compiler cannot compile the program file, and the SQL executor cannot execute it.

For more information on the Binder program, see the *Binder Manual*. For more information on binding HP COBOL programs, see the *COBOL85 for NonStop Systems Manual*.

The nld or ld Utility

The process of linking by the `nld` or `ld` utility applies only to native TNS/R objects created by the NMCOBOL compiler (or by the `nmcobol` utility in the OSS environment).

If you specify the RUNNABLE directive when compiling a source program in the Guardian environment, the NMCOBOL compiler uses the COBOLFE process and a linker to validate and resolve internal and external references and produce an executable object file. If you do not specify the RUNNABLE directive when compiling the source program, you must invoke a linker after compilation to link the object files.

△ **Caution.** The -strip option of the linker removes symbols information from an object file. Without the symbols region, the SQL compiler cannot compile the program file, and the SQL executor cannot execute it.

Use either the `nld` or `ld` utility to link TNS/R object files:

- If you compiled the source program with the NON-SHARED directive (the default), use the `nld` utility to produce a non-PIC loadfile or linkfile.

  To run the `nld` utility in the Guardian environment, enter `nld` at the TACL prompt. In this example, the `nld` utility links the object files to create a non-PIC loadfile, `myprog`:

  ```
  nld embdsqlobj obja objb -o myprog
  ```

  For more information on the `nld` utility, see the *nld and noft Manual*.

- If you compiled the source program with the CALL-SHARED directive, use the `ld` utility to produce a PIC loadfile or linkfile.

  To run the `ld` utility in the Guardian environment, enter `ld` at the TACL prompt. In this example, the `ld` utility links the object files to create a PIC loadfile, `myprog`:

  ```
  ld embdsqlobj obja objb -o myprog
  ```

  For more information on the `ld` utility, see the *ld and rld Reference Manual*.

For more information on linking HP COBOL programs, see the *COBOL85 for NonStop Systems Manual*.

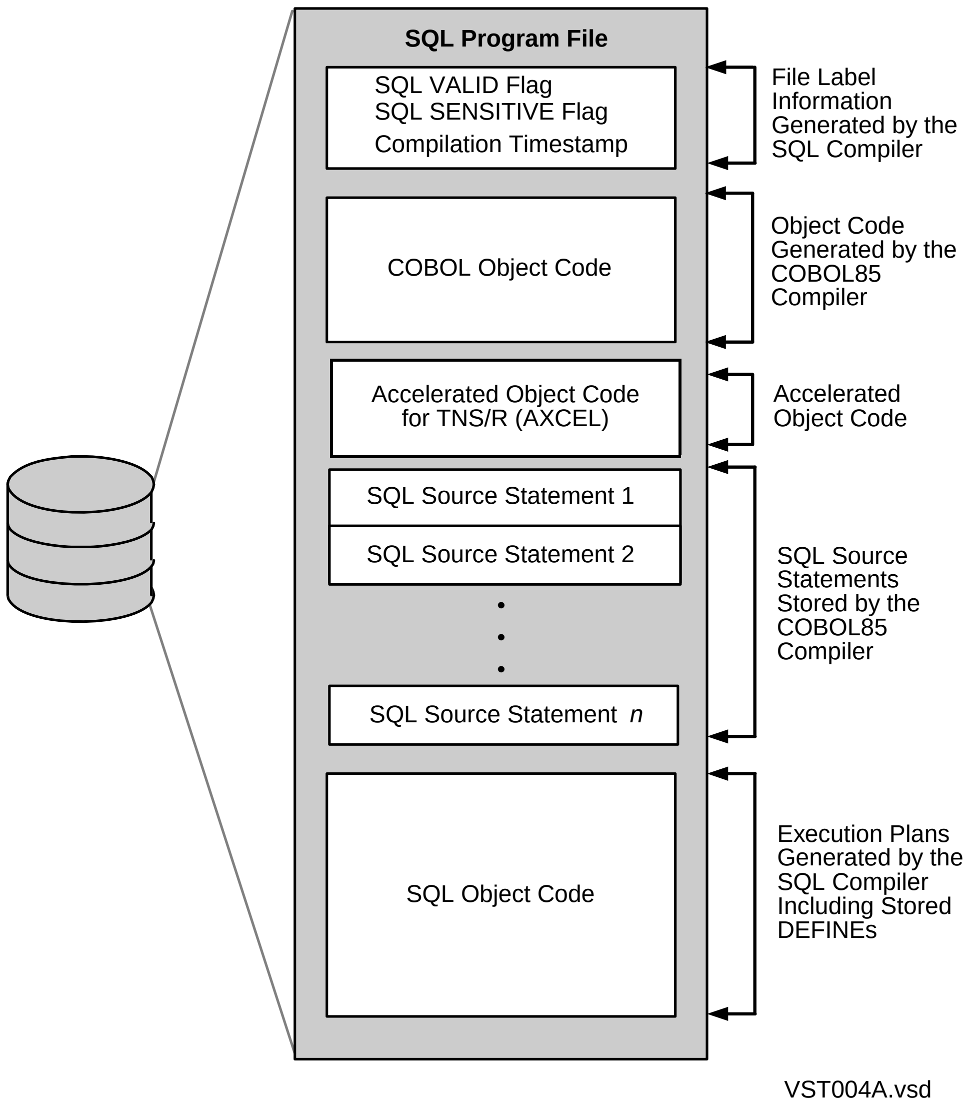
Acceleration of TNS HP COBOL Programs

The process of acceleration applies only to object files created in TNS mode by the COBOL85 compiler (or by the `cobol` utility in the OSS environment of a TNS/R system). Natively compiled programs cannot be accelerated. Accelerated object code improves the execution speed of TNS programs on TNS/R systems. To accelerate a TNS object file, use The Accelerator on page 6-24.

Figure 6-3 shows an SQL program file that has accelerated object code.

△ **Caution.** Because the Accelerator invalidates SQL program files, run the Accelerator before you explicitly SQL compile the program to avoid having to recompile.

Figure 6-3. Accelerated SQL Program File


SQL Program File
SQL VALID Flag
SQL SENSITIVE Flag
Compilation Timestamp
COBOL Object Code
Accelerated Object Code for TNS/R (AXCEL)
SQL Source Statement 1
SQL Source Statement 2
SQL Source Statement n
SQL Object Code
File Label Information Generated by the SQL Compiler
Object Code Generated by the COBOL85 Compiler
Accelerated Object Code
SQL Source Statements Stored by the COBOL85 Compiler
Execution Plans Generated by the SQL Compiler Including Stored DEFINEs
VST004A.vsd

The Accelerator

The Accelerator enables you to optimize TNS programs to run faster on TNS/R systems. A TNS object file that has been accelerated for a TNS/R system has the original TNS code plus the logically equivalent optimized RISC instructions.

If you compiled a TNS object file in the Guardian environment or chose not to accelerate a TNS object file during compilation in the OSS environment, run the Accelerator at a TACL prompt to accelerate the object file.

Note. In the OSS environment of a TNS/R system, you can invoke the Accelerator by specifying the `-Waxcel` flag on the `cobol` utility command line. For more information, see Running the cobol Utility in the OSS Environment on page 6-17.

For example, this command accelerates a TNS object file, `embdsqlobj`, which contains embedded SQL/MP statements, and generates an accelerated object file named `tnsrsqlobj`:

```
AXCEL embdsqlobj, tnsrsqlobj
```

For more information, see the *Accelerator Manual*.

Running the SQL Compiler

The SQL compiler (SQLCOMP) generates SQL object code in the program file. SQLCOMP verifies SQL objects used in SQL statements and generates an optimized execution plan for each SQL statement. Optionally, you can invoke the EXPLAIN utility during compilation to generate a report on the execution plans for SQL DML statements and DEFINEs used by the program.

△ **Caution.** You can use the Accelerator to optimize TNS object code running on a TNS/R system. The Accelerator, however, invalidates SQL program files. Therefore, run the Accelerator before you explicitly SQL compile the program to avoid having to recompile. For more information, see Acceleration of TNS HP COBOL Programs on page 6-23.

Required Access Authority

To run the SQL compiler for an SQL program file, you must have this access authority:

- Read and purge access to the SQL program file
- Read and write access to the PROGRAMS, USAGES, and TRANSIDS tables of the catalog in which the SQL program file is to be registered
- Read and write access to the USAGES and TRANSIDS tables of any catalog in which a table, view, collation, or index that the SQL program file uses is registered

SQL Compiler Functions

- Resolves names and expands SQL object names, including DEFINE names, using the current default volume and the current catalog, and then stores the DEFINE names in the SQL object file.
- Performs type checking for COBOL and SQL data types.
- Expands views.
- Checks object references in catalogs for SQL object names to verify their existence and to read their descriptions, and then evaluates the object type and characteristics for each reference.
- Determines an optimized execution plan by analyzing SELECT, INSERT, UPDATE, and DELETE statements to determine the best access paths and join, sort, and blocking strategies. Estimates the execution costs for DML statements based on the statistics in the catalogs.
- Generates executable code for the execution plans.
- Registers the program in the specified PROGRAMS table and stores any dependencies for tables, views, collations, and indexes in the USAGES table for each table, view, or index that is accessed.

  For example, if a program refers to a collation, SQLCOMP generates a row in the USAGES table showing that the program depends on the collation and sets USAGES.USEDOBJTYPE to CP.
- Generates a listing of the SQL statements in the program file, including warning or error messages that occur.
- Sets the SQL SENSITIVE and SQL VALID flags in the program file label if the compilation is successful.

Running the SQL Compiler in the Guardian Environment

To run the SQL compiler in the Guardian environment, enter the SQLCOMP command at the TACL prompt or from a TACL OBEY command file using this syntax:

```
SQLCOMP / IN object-file [ , OUT [ list-file ] ]

          [ , run-option] [ , run-option ]... /

          [ compiler-option [ , compiler-option ]... ]

compiler-option is:

   [ CATALOG catalog-name                         ]
   [ CURRENTDEFINES | STOREDDEFINES               ]
   [ EXPLAIN                                      ]
   [    [ PLAN ]                                  ]
   [    [ DEFINES [ file-name ] [, OBEYFORM ] ]   ]
   [                                              ]
   [ NOEXPLAIN                                    ]

   [ FORCE                | NOFORCE      ]
   [ OBJECT               | NOOBJECT     ]
   [ RECOMPILE            | NORECOMPILE  ]
   [ RECOMPILEONDEMAND    | RECOMPILEALL ]
   [ REGISTERONLY { ON    | OFF }        ]
   [ NOREGISTER   { ON    | OFF }        ]

   [ CHECK { INVALID PROGRAM  }
           { INVALID PLANS    }
           { INOPERABLE PLANS } ]

   [ COMPILE { PROGRAM [ STORE SIMILARITY INFO ] }
             { INVALID PLANS                     }
             { INOPERABLE PLANS                  } ]
```

object-file

is a Guardian disk file name. This file cannot be part of a user library, a system library, or a DLL. The object file can be generated by these programs:

- HP COBOL compiler
- Binder program or linker
- Accelerator
- SQL compiler

You must run the SQL compiler on the same system where *object-file* exists. If you do not specify a system or volume name, the SQL compiler uses current default values.

list-file

is the destination to which the SQL compiler directs the listing. *list-file* can be a disk file name, process name (including a spooler collector), or a device name (including a terminal, magnetic tape unit, or line printer):

[\node.]*file*

\node

is an optional node (system) name.

\file

is one of these Guardian names:

```
[$volume-name.][subvolume-name.]disk-file-name
$device-name
$device-number
$process-name
$spooler-collector-name[.#spooler-location-name]
```

list-file can also be a class SPOOL DEFINE name.

If *list-file* does not exist, the SQL compiler creates it. If *list-file* already exists, the SQL compiler appends the new output to it.

If you specify OUT but omit *list-file*, the SQL compiler does not generate a listing. If you omit OUT, the SQL compiler directs the listing to the OUT file of the invoking process (usually, your home terminal).

run-option

is a TACL RUN command option as described in the *TACL Reference Manual*.

SQL Compiler Options

CATALOG *catalog-name*

is the name of the catalog to hold a description of the program. *catalog-name* is a subvolume name. If you partially qualify the catalog name, the system expands the name by using the current default values.

The SQL compiler, the object file, and the catalog must reside on the same system.

You can also specify a CLASS catalog DEFINE name for *catalog-name*.

If the program was previously SQL compiled and recorded in a different catalog, the *catalog-name* overrides the catalog name stored in the program file. The program is dropped from the previous catalog and recorded in *catalog-name*.

If you omit the CATALOG clause, the SQL compiler uses the current default catalog. If you have not defined a default catalog, the SQL compiler uses your current default subvolume.

`CURRENTDEFINES | STOREDDEFINES`

specifies the set of DEFINEs used to interpret DEFINE names in the SQL statements in the program file.

`CURRENTDEFINES`

selects the current set of DEFINEs for compiling the program. CURRENTDEFINES is the default.

`STOREDDEFINES`

selects the set of DEFINEs stored with the program the last time it was SQL compiled. This option applies only to previously compiled SQL programs.

`EXPLAIN`

invokes the EXPLAIN utility.

`PLAN`

selects the optimized execution plan determined by the SQL compiler for DML statements in the program. PLAN is the default.

`DEFINES [` *`file-name`* `] [ , OBEYFORM ]`

generates a listing of the TACL DEFINEs that the SQL compiler used to compile the SQL statements. (The SQL compiler uses these DEFINEs to recompile the program if you specify the STOREDDEFINES option.)

`file-name`

is the destination to which the DEFINE listing is written in addition to the compiler listing. See *`list-file`* for a description.

`OBEYFORM`

directs the SQL compiler to write the DEFINE listing in an OBEY command-file format so that you can use an OBEY command to set the DEFINEs. If you omit OBEYFORM, the SQL compiler uses the format displayed by the TACL INFO DEFINE command. If you omit DEFINES, the SQL compiler does not generate a DEFINE listing.

`NOEXPLAIN`

disables the EXPLAIN utility. NOEXPLAIN is the default.

`FORCE | NOFORCE`

controls how errors affect SQL compilation.

`FORCE`

directs the SQL compiler to produce a valid, executable object file regardless of syntax errors. The SQL compiler writes the SQL source statements to the program file so that the statements can automatically be recompiled if executed at run time. Use the FORCE option to debug a program if you do not need to execute the SQL statements that generate errors.

`NOFORCE`

directs the SQL compiler to produce the SQL object code only if there are no syntax errors. NOFORCE is the default.

`OBJECT | NOOBJECT`

controls whether the compiler produces an SQL program file.

`OBJECT`

directs the compiler to produce a program file (depending on whether errors occur and whether the FORCE or NOFORCE option is in effect). OBJECT is the default.

`NOOBJECT`

directs the compiler to perform checking functions and to generate an EXPLAIN listing, if requested, but not to produce a program file.

`RECOMPILE | NORECOMPILE`

specifies whether the program should be automatically recompiled, if necessary, during program execution.

`RECOMPILE`

directs the SQL executor to automatically recompile a program whenever any of these conditions occur:

- The program file is SQL invalid.
- The DEFINEs used at SQL load time are different from the DEFINEs used during explicit SQL compilation.
- The timestamp check fails for an SQL object in an SQL statement.
- An access path (index) is unavailable.

RECOMPILE is the default.

If the program uses the similarity check, automatic recompilation might not occur. For more information, see Section 8, Program Invalidation and Automatic SQL Recompilation.

`NORECOMPILE`

directs the SQL executor not to automatically recompile the program. If any of the conditions described under the RECOMPILE option occur during execution, an error is generated, and the program is subject to explicit SQL recompilation for validation.

`RECOMPILEONDEMAND | RECOMPILEALL`

specifies whether the SQL executor should invoke the recompilation of an entire invalid program or only SQL statements actually executed. If you specify NORECOMPILE, this option is ignored.

`RECOMPILEONDEMAND`

directs the SQL executor to recompile only the statements in the invalid program that are actually executed. Automatic recompilation occurs the first time the individual SQL statement is executed.

`RECOMPILEALL`

directs the SQL executor to automatically recompile the entire program at SQL load time if it is invalid. RECOMPILEALL is the default.

`REGISTERONLY`

directs the SQL compiler to register a previously SQL compiled program in a specific catalog without recompiling the program. To use the REGISTERONLY option, you must have an SQL/MP software version of 310 (or later).

`ON`

directs the SQL compiler to register a program in the specified catalog without compiling the SQL statements in the program or creating a new program file. The SQL compiler marks the program’s file label as SQL sensitive and SQL valid. The program retains its existing execution plans. If the program was not previously SQL compiled, the operation fails with SQL error 2115.

The CATALOG option is the only other SQLCOMP option you can specify with the REGISTERONLY ON option. If you specify an option other than CATALOG, the operation fails with SQL error 2111. If the program was previously compiled with the NOREGISTER ON option, the operation fails with SQL error 2108. If the program was modified by the Binder program or linker after it was SQL compiled, the operation fails with SQL error 2103.

`OFF`

directs the SQL compiler to explicitly SQL compile the program and perform all SQL compiler functions.

OFF is the default.

`NOREGISTER`

directs the SQL compiler to compile a program without registering the program in a catalog. To use the NOREGISTER option, you must have an SQL/MP software version of 310 (or later).

`ON`

directs the SQL compiler to explicitly compile the program but not to register it in a catalog. The SQL compiler does not mark the program as SQL sensitive and SQL valid in its file label. Therefore, the program file can be executed without being registered in an SQL catalog. If you specify the CATALOG option with the NOREGISTER ON option, the compilation fails with SQL error 2116. If the program is already registered in a catalog, the compilation fails with SQL error 2110. If the program was modified by the Binder program or linker after it was SQL compiled, the operation fails with SQL error 2103.

`OFF`

directs the SQL compiler to explicitly compile the program and perform all specified compiler functions, including registering the program in the catalog.

OFF is the default.

`CHECK`

determines the behavior of the SQL executor when it executes an invalid SQL statement or a statement that references a DEFINE that has changed since the last explicit SQL compilation. To use a CHECK option, you must have an SQL/MP software version of 310 (or later).

If you specify the CHECK INVALID PLANS or CHECK INOPERABLE PLANS option (which stores similarity information in the program file), the SQL compiler sets the program’s PFV and PCV to 310 (or later). To support these options, an SQL catalog must have a catalog version of 310 (or later).

If you restore a program using the SQLCOMPILE option, the RESTORE program invokes the recompilation of the program by using the SQLCOMP CHECK option specified during the last explicit SQL compilation.

`INVALID PROGRAM`

specifies that the SQL executor should automatically recompile all SQL statements in an invalid program or a program that references changed

DEFINEs (if NORECOMPILE is not specified). The SQL executor does not attempt to execute any plans in the program without recompiling them.

CHECK INVALID PROGRAM is the default.

`INVALID PLANS`

specifies that the SQL executor should automatically recompile an SQL statement if either of these conditions occur (and NORECOMPILE is not specified):

- The statement is invalid. Invalid statements have plans that fail the redefinition timestamp.
- The statement references a DEFINE at SQL load time that has changed since the last explicit SQL compilation.

The SQL executor reuses the execution plans from the program file for the other SQL statements, which have valid plans and unchanged DEFINE values.

During explicit SQL compilation, the CHECK INVALID PLANS option directs the SQL compiler to store similarity information in the program file (even if the similarity check is not enabled for the table or protection view).

`INOPERABLE PLANS`

specifies that the SQL executor should perform the similarity check on each SQL object in an SQL statement if the similarity check is enabled for referenced tables and protection views and either of these conditions occur:

- The statement is invalid. Invalid statements have plans that fail the redefinition timestamp.
- The statement references a DEFINE at SQL load time that has changed since the last explicit SQL compilation.

If the similarity check passes, the SQL executor considers the plan to be operable (although it might not be optimal) and executes the SQL statement without automatically recompiling it, therefore reusing the existing execution plan.

If the similarity check fails, the SQL executor considers the plan to be inoperable. The SQL executor then recompiles (in memory only) the SQL statement that generated the inoperable plan (if NORECOMPILE is not specified) and executes the recompiled statement.

During explicit SQL compilation, the CHECK INOPERABLE PLANS option directs the SQL compiler to store similarity information in the program file (even if the similarity check is not enabled for the table or protection view).

`COMPILE`

determines which SQL statements are compiled during an explicit SQL compilation. You can direct the SQL compiler to use the similarity check to determine if a statement’s execution plan from a previous compilation is operable. The SQL compiler then recompiles only the statements that fail the similarity check. Other SQL statements retain their existing plans.

To use a COMPILE option, you must have an SQL/MP software version of 310 (or later).

To support the COMPILE PROGRAM STORE SIMILARITY INFO, COMPILE INVALID PLANS, or COMPILE INOPERABLE PLANS option, an SQL catalog must have a catalog version of 310 (or later).

If you specify the COMPILE PROGRAM STORE SIMILARITY INFO, COMPILE INVALID PLANS, or COMPILE INOPERABLE PLANS option (which stores similarity information in the program file), the SQL compiler sets the program’s PFV and PCV to 310. If you omit the COMPILE option or specify the COMPILE PROGRAM option (the default), the SQL compiler sets the PCV to 1 (unless the program includes other version 310 features).

`PROGRAM [ STORE SIMILARITY INFO ]`

directs the SQL compiler to explicitly compile all SQL statements in the program. If you include the STORE SIMILARITY INFO clause, the SQL compiler also stores similarity information for each SQL statement in the program file.

CHECK PROGRAM is the default.

`INVALID PLANS`

directs the SQL compiler to explicitly compile these SQL statements:

- Statements that reference changed DEFINEs.
- Statements with plans that fail the redefinition timestamp check.
- Statements with altered execution plans, which are invalid but operable plans that the SQL compiler has updated without recompiling.
- Uncompiled SQL statements with empty sections. The SQL compiler generates an empty section if an SQL statement references a nonexistent DEFINE or SQL object.

Other SQL statements retain their existing execution plans.

The COMPILE INVALID PLANS option stores similarity information in the program file and updates the program’s dependencies on database objects in the USAGES tables.

If the program has not been previously compiled or if the program does not contain similarity information, the COMPILE INVALID PLANS option directs the SQL compiler to compile all SQL statements in the program.

`INOPERABLE PLANS`

directs the SQL compiler to explicitly compile these SQL statements:

- Statements with inoperable plans (invalid plans that fail the similarity check).
- Uncompiled statements with empty sections. The SQL compiler generates an empty section if an SQL statement references a nonexistent DEFINE or SQL object. (The SQL compiler also generates empty sections for CONTROL directives and DDL statements.)

Other SQL statements retain their existing execution plans.

The COMPILE INOPERABLE PLANS option stores similarity information in the program file and updates the program's name map and usages in the USAGES tables.

If the program has not been previously compiled or if the program does not contain similarity information, the COMPILE INOPERABLE PLANS option directs the SQL compiler to compile all SQL statements in the program.

To terminate a compilation in the Guardian environment, use the Break key to return to the Command Interpreter and stop the process using its process identification number.

Running the SQL Compiler in the OSS Environment

In the OSS environment, the SQL compiler is invoked by the `cobol` utility (on a TNS/R system) with the `-Wsql` flag or by the `nmcobol` utility with the `-Wsql` or `-Wsqlcomp` flag. For more information, see Running HP COBOL Compilers in the OSS Environment on page 6-16 or the *Open System Services Shell and Utilities Reference Manual*.

Using Current Statistics

For the SQL compiler to generate the best execution plan, it must have current statistics for the referenced tables. SQL/MP does not automatically update these statistics. A program must execute the UPDATE STATISTICS statement to generate current statistics in a catalog.

To execute the UPDATE STATISTICS statement, the process started by the program must:

- Have read access to the table
- Have write access to the catalogs that contain the table descriptions

- Be the local owner of the table or a remote owner with purge access to the table (or be the local super ID user)

For information about process access, see Required Access Authority on page 7-1.

In these examples, the first statement updates the statistics for all columns in the ORDERS table. The second statement updates the statistics columns in the primary key or clustering key or in any indexes for the table ODETAIL.

```
EXEC SQL UPDATE ALL STATISTICS FOR TABLE =ORDERS END-EXEC.
EXEC SQL UPDATE STATISTICS FOR TABLE =ODETAIL END-EXEC.
```

For more information about the UPDATE STATISTICS statement, see the *SQL/MP Reference Manual*.

SQL Compiler Messages

The SQL compiler issues messages for error and warning conditions. An error can prevent successful compilation of a program file, but a warning does not. For a description of all SQL compiler messages, see the *SQL/MP Messages Manual*.

Error Conditions

An error condition results from an invalid reference to an SQL object in an SQL statement. Examples of invalid references are an incorrect column name or an incompatible data type. If an error occurs, the SQL compiler generates a listing, but it does not record the program file in the catalog and does not validate it for execution.

You can force an SQL compilation regardless of errors by specifying the SQLCOMP FORCE option. The FORCE option directs the compiler to record the SQL program file in the catalog and to validate it for execution even if errors occur. The SQL compiler also writes the SQL statements with errors to the program file so that the statements can be automatically recompiled at run time. You can use the FORCE option to debug a program when you are not concerned about executing the SQL statements that produce errors.

Dynamic SQL statements are not compiled during explicit SQL compilation. Errors in these statements are returned at run time after dynamic compilation by a PREPARE or EXECUTE IMMEDIATE statement.

Warning Conditions

A warning condition usually occurs when the SQL compiler has insufficient information available. If a warning occurs, the SQL compiler still records the program file in the catalog, validates the file for execution, and then returns a warning message.

In these two situations, the SQL compiler issues a warning message but still compiles the statement:

- Compiler assumption. The SQL compiler made an assumption necessary to complete the compilation. For example, if the number of columns in the SELECT statement does not match the number of host variables, the compiler returns a warning message and assumes that you do not want to use either the extra columns or the extra host variables.

- Unavailable statistics. The SQL compiler does not have the necessary statistics for a table or view to optimize an execution plan. The compiler then uses statistics in the catalog to determine an optimized execution plan.

In other situations, the SQL compiler marks the statement as having insufficient information to compile and does not record dependencies in the USAGES catalog tables for the affected statement. The SQL executor then tries to resolve the problem at run time by automatically recompiling the statement.

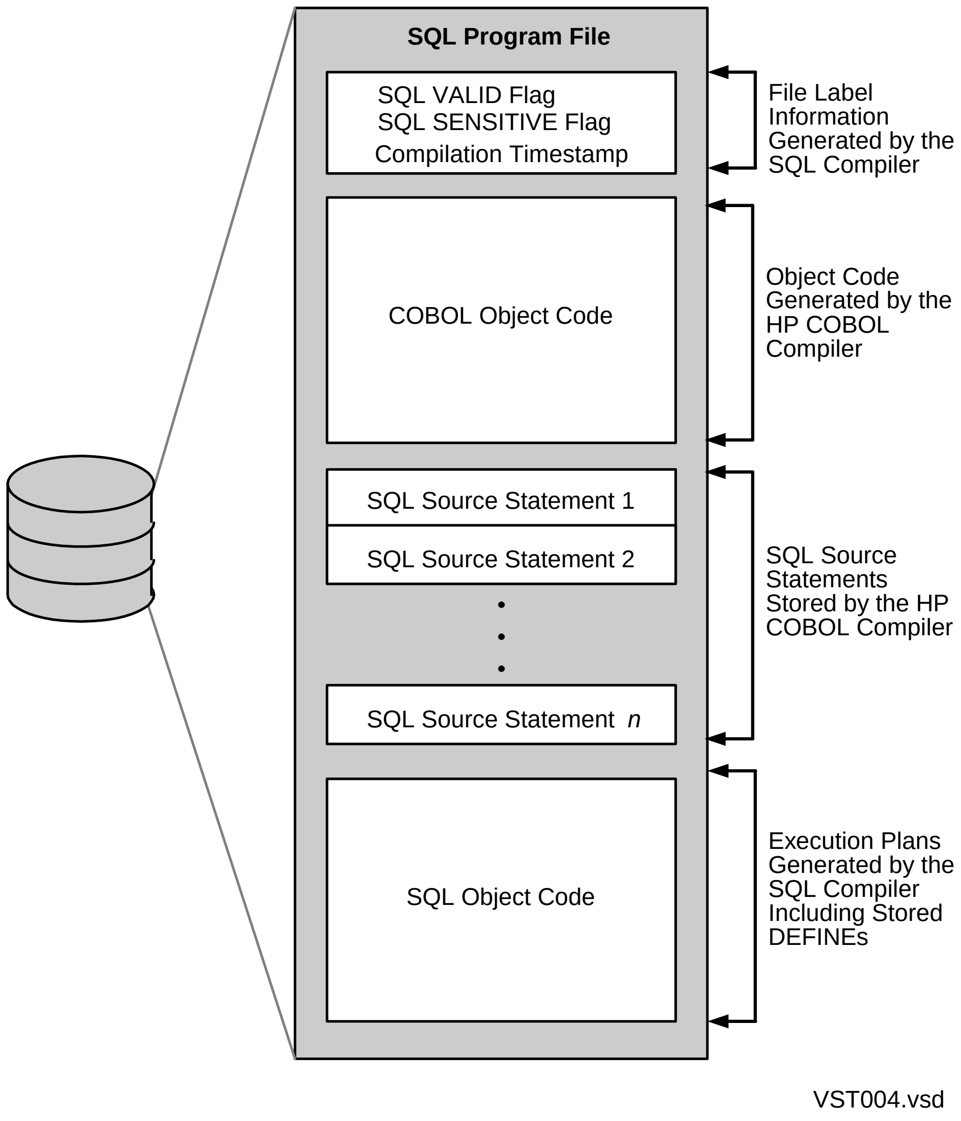
At run time, the uncompiled statement causes an error in these cases:

- Insufficient information. The SQL compiler does not have enough information to determine the validity of a statement. For example, an SQL statement refers to an unavailable table. The table might not exist, or it might reside on an unavailable remote node. (This situation always occurs for a program that both creates and refers to a table. The table, of course, does not exist when the program is explicitly SQL compiled.)
- Unresolved DEFINEs. An SQL statement references a nonexistent DEFINE.

SQL Program File Format

The input program file to the SQL compiler can be a COBOL object file, a file generated by the Binder program (TNS programs) or linker (native programs), a file generated by the Accelerator (TNS programs only), or a file previously compiled by the SQL compiler. Figure 6-4 shows the format of an SQL program file. For an SQL program file that has accelerated object code, see Figure 6-3 on page 6-24.

Figure 6-4. SQL Program File Format


SQL Program File
SQL VALID Flag
SQL SENSITIVE Flag
Compilation Timestamp
COBOL Object Code
SQL Source Statement 1
SQL Source Statement 2
SQL Source Statement *n*
SQL Object Code
File Label Information Generated by the SQL Compiler
Object Code Generated by the HP COBOL Compiler
SQL Source Statements Stored by the HP COBOL Compiler
Execution Plans Generated by the SQL Compiler Including Stored DEFINEs
VST004.vsd

SQL Compiler Listings

The SQL compiler writes all SQL statements in the program file to the listing (or OUT) file. If an error or warning occurs, the compiler includes a message after the statement that caused the problem. For DML statements, the compiler also includes the estimated cost of processing the statement, which is a positive integer indicating the relative cost. The larger the integer, the more CPU time and disk access time are required to execute the statement. Example 6-1 is a sample compiler listing.

Example 6-1. Sample SQL Compiler Listing of a COBOL Program (page 1 of 2)

```
 SQL -  Source File = \NEWYORK.$SQL.SQLPGMS.COBPGM

 SQL -  SLT Index   = 0,   Run-Unit  = NEWPART

30                    DECLARE GET_SUPPLIER_CURSOR CURSOR FOR
31                    SELECT SUPPNUM,
32                           SUPPNAME,
33                           STREET,
34                           CITY,
35                           STATE,
36                           POSTCODE
37                    FROM   =SUPPLIER
38                    WHERE  SUPPNUM  = :SUPPLIER-OF-PARTS
39                    REPEATABLE ACCESS
40
*** Statistics: Estimated cost: 1

 SQL -  SLT Index   = 1,   Run-Unit  = NEWPART

73                    BEGIN WORK

 SQL -  SLT Index   = 2,   Run-Unit  = NEWPART

97                      INSERT INTO =PARTLOC
98                         VALUES (:LOC-CODE    OF PARTLOC-REC,
99                                 :PARTNUM     OF PARTLOC-REC,
100                                :QTY-ON-HAND OF PARTLOC-REC)
101
*** Statistics: Estimated cost: 1

 SQL -  SLT Index   = 3,   Run-Unit  = NEWPART

110                      INSERT INTO =PARTS
111                         VALUES (:PARTNUM        OF PARTS-REC,
112                                 :PARTDESC       OF PARTS-REC,
113                                 :PRICE          OF PARTS-REC,
114                                 :QTY-AVAILABLE  OF PARTS-REC)
115
*** Statistics: Estimated cost: 2
```

Example 6-1. Sample SQL Compiler Listing of a COBOL Program (page 2 of 2)

```
 SQL -  SLT Index   = 4,   Run-Unit  = NEWPART

121                    COMMIT WORK

 SQL -  SLT Index   = 5,   Run-Unit  = NEWPART

137                     ROLLBACK WORK
BINDER - OBJECT FILE BINDER - T9621D30 - (17JUL95)   SYSTEM \NEWYORK
Copyright Tandem Computers Incorporated 1982-1995

Object file \NEWYORK.$SQL.SQLPGMS.COBOBJ
TIMESTAMP  1996-06-17 14:47:05

         0  Binder Warnings
         0  Binder Errors

 SQL **************************************************
 SQL - Summary of SQL Compiling
 SQL -   Number of SQL Statements = 6
 SQL -   Number of SQL Errors     = 0
 SQL -   Number of SQL Warnings   = 0
 SQL -   Number of other Errors   = 0
 SQL -   Compile Time             = 00:00:00.249
 SQL -   Elapsed Time             = 00:00:22.915
 SQL -   Program file is \NEWYORK.$SQL.SQLPGMS.COBOBJ
 SQL -   >>> SQL COMPILATION STORED IN PROGRAM FILE <<<
 SQL **************************************************
```

Using the EXPLAIN Utility

The EXPLAIN utility generates reports about execution plans for each SQL statement. Use EXPLAIN reports to determine the tables and indexes used by a program and whether creating other indexes or modifying a query would improve the performance of the program. The EXPLAIN utility has these report options:

- EXPLAIN PLAN report
- EXPLAIN DEFINES report

EXPLAIN PLAN Report

The EXPLAIN PLAN report, which applies only to DML statements, shows the strategy for executing a DML statement and includes optimized access paths, joins, and sorts. The EXPLAIN PLAN report generates a plan for a statement containing subqueries in separate query plans: one for each subquery and one for the statement itself. This report numbers the query plans in each statement in the order they appear. Each plan can contain these steps:

- Scan a table
- Join two or more tables
- Insert into a table
- Perform a sort operation

In this example, the SQL compiler compiles the SQLPROG program file using the EXPLAIN PLAN option. The SQLCOMP command specifies a catalog other than the current default catalog. The SQL compiler uses the current set of DEFINEs and writes the output to the spooler location $S.#EXPLAIN:

```
SQLCOMP / IN SQLPROG, OUT $S.#EXPLAIN /
CATALOG $DISK2.SALES, EXPLAIN PLAN
```

EXPLAIN DEFINES Report

The EXPLAIN DEFINES report shows the mapping of DEFINE names used in SQL statements with this information:

- The default volume and catalog used by the program (obtained from the =_DEFAULTS DEFINE)
- Each DEFINE name and its associated Guardian name

The EXPLAIN utility can generate EXPLAIN DEFINES reports in either of these formats:

OBEY command file format	EXPLAIN generates the ADD DEFINE commands that add DEFINEs. You can then use a TACL OBEY command to execute these commands.
INFO DEFINE format	EXPLAIN generates a report in the format used by the TACL INFO DEFINE command.

This example shows an OBEY command file format report. In an actual report, each instance of *subvolume-name*, *guardian-name*, and *define-name* would be replaced by the actual name.

```
ALTER DEFINE =_DEFAULTS, VOLUME  subvolume-name
ALTER DEFINE =_DEFAULTS, CATALOG subvolume-name

ADD DEFINE   define-name, FILE guardian-name
ADD DEFINE   define-name, FILE guardian-name
...
```

When you issue an OBEY command to execute the file shown previously, ensure that the DEFINE mode (DEFMODE) is ON, and the DEFINE CLASS is MAP.

The INFO DEFINE format uses the same format as the INFO DEFINE command. This example shows an INFO DEFINE format report. In an actual report, each *guardian-name* and *define-name* would be replaced by the actual name.

```
DEFINE NAME          =_DEFAULTS
CLASS                DEFAULTS
VOLUME               guardian-name
CATALOG              guardian-name

DEFINE NAME          define-name
CLASS                MAP
FILE                 guardian-name

DEFINE NAME          define-name
CLASS                MAP
```

```
FILE                    guardian-name
...                     ...
```

In the next example, the SQL compiler writes an execution plan and DEFINEs to the spooler location $S.#EXPLAIN. The OBEYFORM option directs the compiler to write the DEFINEs in OBEY command file format to the file named SETDEFS for subsequent execution. The catalog name is not included in the SQLCOMP command because it is stored in the program file. The NOOBJECT option suppresses the generation of a new object file, so the SQL compiler does not register the program file in a catalog.

```
SQLCOMP /IN SQLPROG,OUT $S.#EXPLAIN/ NOOBJECT,
   EXPLAIN PLAN DEFINES SETDEFS, OBEYFORM
```

For more information about the EXPLAIN utility, including detailed examples and reports, see the *SQL/MP Query Guide*.

Using CONTROL Directives

You can use these CONTROL directives with either static or dynamic SQL statements in a COBOL program:

`CONTROL EXECUTOR`

allows or prohibits parallel execution of a query by multiple SQL executors. Parallel execution can decrease the elapsed time for processing a query.

`CONTROL QUERY`

controls query execution plans as follows:

- Optimization of query response time for returning only the first few rows found or for returning all rows found
- Use of hash join algorithms in execution plans
- Use of execution-time name resolution to resolve names in execution plans when the SQL statement executes rather than during explicit SQL compilation or at SQL load time

`CONTROL TABLE`

controls these performance-related options for accessing tables and views:

- Selection of access paths, join methods, join sequences, and lock types
- Selection of block buffering and block splitting algorithms
- Action to take for locked data or unavailable partitions
- Opening of indexes and partitions at the initial access to a table
- Checkpointing of unaudited write operations

The use of CONTROL directives in a COBOL program is discussed next. For the syntax of each CONTROL directive, see the *SQL/MP Reference Manual*.

Static SQL Statements

Follow these guidelines when you use CONTROL directives with static SQL statements:

- A CONTROL directive affects subsequent static DML statements in listing order, regardless of the execution order, until either of these conditions occur:
    - Another CONTROL directive resets the CONTROL options.
    - The program encounters the end of the run-time data unit (RTDU) that contains the CONTROL directive. (An RTDU is a region of the program file that contains both SQL source statements and object code.)

  Each COBOL main program and nested program is a separate RTDU. Therefore, a CONTROL directive in a main program does not affect statements in a nested program, and a CONTROL directive in a nested program affects only statements in the nested program but not in the main program or in other nested programs.

  An SQL map shows each RTDU. To generate an SQL map in the compiler listing, specify the SQLMAP option in the SQL directive.
- A dynamic CONTROL directive does not affect static SQL statements in the program, except as described in the note under Dynamic SQL Statements on page 6-44.
- A CONTROL directive coded within flow-control statements (for example, IF and ELSE) affects SQL statements in the listing order regardless of the execution order.
- To affect a cursor, you must code the CONTROL directive before the DECLARE CURSOR statement. The CONTROL directive must also be in the same RTDU as the DECLARE CURSOR statement.

In this example, the CONTROL EXECUTOR directive specifies parallel evaluation when the program executes the first FETCH statement for the cursor:

```
EXEC SQL
  CONTROL EXECUTOR PARALLEL EXECUTION ON END-EXEC.
EXEC SQL
  DECLARE LIST_CUSTOMERS_WITH_ORDERS CURSOR FOR
     SELECT CUSTOMER.CUSTNUM ,
            CUSTOMER.CUSTNAME
     FROM  =CUSTOMER, =ORDERS
     WHERE  CUSTOMER.CUSTNUM = ORDERS.CUSTNUM
     STABLE ACCESS END-EXEC.
```

This example varies the wait time for cursors that access the PARTS table. The default wait time (60 seconds) applies only to the first cursor (CURSOR1):

```
PROCEDURE DIVISION.
...
200-DEFAULT-WAIT.
EXEC SQL
  DECLARE CURSOR CURSOR1 FOR SELECT PARTNUM, PARTDESC, PRICE
  FROM SALES.PARTS
  WHERE (PARTNUM > :MIN-PARTNUM AND PARTNUM < :MAX-PARTNUM)
  ORDER BY PARTNUM END-EXEC.
...

...
500-SHORT-WAIT.
EXEC SQL
  CONTROL TABLE SALES.PARTS TIMEOUT .1 SECOND END-EXEC.

EXEC SQL
  DECLARE CURSOR CURSOR2 FOR SELECT PARTNUM, PARTDESC, PRICE
  FROM SALES.PARTS
  WHERE (PARTNUM > :MIN-PARTNUM AND PARTNUM < :MAX-PARTNUM)
  ORDER BY PARTNUM END-EXEC.

800-INFINITE-WAIT.
EXEC SQL
  CONTROL TABLE SALES.PARTS TIMEOUT -1 SECOND END-EXEC.

EXEC SQL
  DECLARE CURSOR CURSOR3 FOR SELECT PARTNUM, PARTDESC, PRICE
  FROM SALES.PARTS
  WHERE (PARTNUM > :MIN-PARTNUM AND PARTNUM < :MAX-PARTNUM)
  ORDER BY PARTNUM END-EXEC.
...
```

Dynamic SQL Statements

A static CONTROL TABLE directive does not affect dynamic SQL statements. To use a CONTROL TABLE directive with dynamic SQL statements, specify a dynamic CONTROL TABLE directive by using the PREPARE and EXECUTE (or EXECUTE IMMEDIATE) statements. A dynamic CONTROL directive affects only dynamic SQL statements prepared after the CONTROL directive in execution order, except as noted.

Note. A dynamic CONTROL TABLE directive with the TIMEOUT option affects all static and dynamic SQL statements that follow in execution order (as opposed to listing order) until another dynamic CONTROL TABLE directive resets the TIMEOUT option or until the program encounters the end of the run-time data unit (RTDU) that contains the CONTROL TABLE directive.

Using Compatible Components

Before you compile an SQL program file, you might need to determine the versions of the HP COBOL compiler, SQL compiler (SQLCOMP), and all SQL program files to ensure that all components and files are compatible.

HP COBOL Compiler

The host SQL version (HSV) identifies the SQL version of the HP COBOL compiler. A COBOL program that uses version 300 (or later) SQL features must be compiled with a version of 300 (or later) HP COBOL compiler. To determine the version of the HP COBOL compiler, use one of these methods:

- Run the VPROC program for the HP COBOL compiler object file. VPROC displays a line for each object bound into the target object file. Check the version in the VPROC line that contains S7094, which is the SQL compiler interface (SCI) product number.
- When you run the HP COBOL compiler, specify the SQLMAP option in the SQL compiler directive. The SQLMAP option directs the HP COBOL compiler to include the HOSV in the map at the end of the source-file listing. For example, a version 335 COBOL compiler listing includes this line:

```
Host Object SQL Version = 335
```

SQL Compiler

The SQL compiler (SQLCOMP) must have the same version as (or later than) the HOSV of the SQL program file. To determine the version of the SQL compiler, use the GET VERSION OF SYSTEM statement. All SQL/MP components on a NonStop system, including the SQL compiler, must have the same version.

SQL Program File

An SQL program file has these versions:

Host object SQL version (HOSV)	The version of the HP COBOL compiler used to compile the program. Generated by the COBOL compiler, the version is, therefore, the same as the host SQL version (HSV) of the compiler.
Program format version (PFV)	The version of the SQL compiler used to compile the program and the oldest version of an SQL executor that can execute the program. Generated by the SQL compiler.
Program catalog version (PCV)	The oldest version of an SQL catalog in which the program can be registered. Generated by the SQL compiler.

This subsection describes the HOSV and its relationship to the HP COBOL compiler and SQL compiler. For more information about the PFV and PCV, see the *SQL/MP Version Management Guide*.

The HP COBOL compiler generates the HOSV and stores the value in the object file.

If multiple object files are bound together in a single target object file, the HOSV of the target object file is the newest (maximum) HOSV of the individual object files. For example, if an object file with an HOSV of 2 and another object file with an HOSV of 310 are bound into a new target object file, the HOSV of the target object file is 310.

To return the HOSV of a program object file, use the GET VERSION OF PROGRAM statement with the HOST OBJECT option. You can execute this statement from SQLCI or embedded in a COBOL program. This GET VERSION OF PROGRAM statement is executed from SQLCI:

```
GET HOST OBJECT VERSION OF PROGRAM sqlprog;

VERSION: 310
--- SQL operation complete.
```

To embed a static GET VERSION OF PROGRAM statement in a COBOL program, you must include the INTO clause with a host variable. This statement returns the HOSV of SQLPROG to the host variable named HV-HOSV:

```
EXEC SQL
  GET HOST OBJECT VERSION OF PROGRAM SQLPROG INTO :HV-HOSV
END-EXEC.
```

You can also execute a dynamic GET VERSION OF PROGRAM statement by using the PREPARE and EXECUTE statements:

```
MOVE "GET HOST OBJECT VERSION OF PROGRAM SQLPROG" TO HV-TEXT.
EXEC SQL
  PREPARE DYNAMIC-STATEMENT FROM :HV-TEXT
END-EXEC.
EXEC SQL
  EXECUTE DYNAMIC-STATEMENT RETURNING :HV-HOSV
END-EXEC.
```

However, you cannot use the GET VERSION OF PROGRAM statement with the EXECUTE IMMEDIATE statement.

For the syntax of the GET VERSION statement, see the *SQL/MP Reference Manual*.

7 Program Execution

This section describes the execution of a COBOL program containing embedded SQL statements and directives in the OSS environment.

Topics include:

Required Access Authority

To execute an SQL program file, you (or the creator process if you use the COBOL CREATEPROCESS routine) must have this access authority:

- Read and execute authority to the SQL program file
- Read authority to the catalog in which the program is registered
- Read authority to any catalogs in which tables or views used by the program are registered for SQL statements that require automatic SQL recompilation

For an embedded SQL statement (either static or dynamic) to access and operate on a database object, such as a table or view, the process started by the program must have specific privileges associated with it. The privileges of both the process access ID (PAID) and group list are evaluated to determine if a process can be granted access to a database object. The group list is always associated with the creator access ID (CAID), which represents the user who starts the process. The PAID depends on the PROGID setting.

If the program owner does not enable the PROGID attribute for the program file, the PAID will be the same as the user ID of the process creator (that is, the CAID). When a user executes the program, the process uses the privileges of the process creator and accesses only resources to which the process creator has access.

If the program owner enables the PROGID attribute for the program file, the PAID will be the same as the user ID of the program owner. When a user executes this program, the process uses the privileges of the program owner and accesses only the resources to which the program owner has access. PROGID programs enable one user to temporarily gain a controlled subset of another user’s privileges. For more information about PROGID programs, see the *Security Management Guide*.

Using DEFINEs

Before running an SQL program file, you can specify DEFINE, PARAM, or ASSIGN commands. This subsection describes DEFINEs. For information about PARAM and ASSIGN commands, see the *TACL Reference Manual*.

You can use DEFINE names in an SQL program to specify the names of SQL catalogs and objects (tables, views, indexes, partitions, and collations). Use a class CATALOG DEFINE for a catalog and a class MAP DEFINE for an object.

You enable and disable DEFINEs by using the DEFMODE attribute. If DEFMODE is ON when a program begins execution, the system propagates the current set of DEFINEs from the process file segment (PFS) of your TACL process to the new process. If DEFMODE is OFF, the system propagates only the =_DEFAULTS DEFINE to the new process. To display the current DEFMODE setting, use the SHOW DEFMODE command.

You can create, modify, delete, and display DEFINEs with TACL (or SQLCI) commands, Guardian system procedures, and OSS shell commands. You can also specify the =_SORT_DEFAULTS DEFINE to control sort operations.

You must set all DEFINE names used in SQL statements before SQL load time unless your program uses execution-time name resolution.

To determine the DEFINE set used when an SQL program was compiled, use the EXPLAIN DEFINES option of the SQLCOMP command.

Entering the TACL RUN Command

To execute an SQL program file from a TACL process, use the TACL RUN (or RUND to invoke the INSPECT program) command. You can enter a RUN command either explicitly or implicitly by using this syntax:

```
[ RUN[D] ] program-file [ / [ , run-option ]... / ]
     [ param-set ]
```

`RUN`

executes the program file without invoking the Inspect debugger.

`RUND`

executes the program file under the control of the Inspect symbolic debugger.

program-file

is the name of the SQL program file. For an explicit RUN command, TACL qualifies a partially qualified file name by using the =_DEFAULTS DEFINE. For an implicit RUN command, TACL searches for *program-file* in the TACL #PMSEARCHLIST variable.

run-option

is a RUN command run option as described in the *TACL Reference Manual*.

param-set

is one or more parameters to pass to the new process.

For example, this RUN command runs the program file, SQLPROG, and specifies the NAME, OUT, and NOWAIT run options:

```
RUN sqlprog / NAME $sqlrun, OUT $s.#sqllist, NOWAIT /
```

This RUND command runs the program file, $DISK2.SQL.SQLPROG, under the control of the Inspect debugger:

```
RUND $disk.sql.sqlprog
```

For additional information about the RUN command, see the *TACL Reference Manual*.

Using the CREATEPROCESS Routine

To execute an SQL program file from a COBOL program, use the COBOL CREATEPROCESS routine. The CREATEPROCESS routine starts a new process by using the parameters you supply and (optionally) sends process-creation messages altered by COBOL saved message utility (SMU) routines to the new process.

To call the CREATEPROCESS routine, use the COBOL ENTER statement. However, do not include the TAL keyword after the ENTER keyword.

Example 7-1 on page 7-4 illustrates a CREATEPROCESS routine. The user enters values for NEW-PROGRAM, NEW-OPTION, and NEW-CPU.

Example 7-1. COBOL CREATEPROCESS Routine

```
IDENTIFICATION DIVISION
 PROGRAM-ID.  RUNSQL.
  ...
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
...
 SPECIAL-NAMES.
 FILE $SYSTEM.SYSTEM.COBOLLIB IS COBOL-LIBRARY.
...
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 SQL-PROGRAM-FILE    PIC X(36).
  01 CREATE-OPTION       PIC 9999  COMP.
  01 PRIMARY-CPU         PIC S9999 COMP.
  01 RETURN-STATUS       PIC S9999 COMP.
...
 PROCEDURE DIVISION.
  ...
* User enters values for NEW-PROGRAM, NEW-OPTION, and NEW-CPU.
 MOVE NEW-PROGRAM TO SQL-PROGRAM-FILE.
 MOVE NEW-OPTION  TO CREATE-OPTION.
 MOVE NEW-CPU     TO PRIMARY-CPU.

 ENTER "CREATEPROCESS" OF COBOL-LIBRARY
        USING SQL-PROGRAM-FILE,
        OMITTED,
        CREATE-OPTION,
        OMITTED,
        PRIMARY-CPU,
        OMITTED,
        OMITTED
        GIVING RETURN-STATUS.

 IF RETURN-STATUS IS NOT EQUAL TO ZERO
      PERFORM 1000-ERROR-ROUTINE.
...
```

For more information about the CREATEPROCESS routine, see the *COBOL85 for NonStop Systems Manual*.

Using the CLU_PROCESS_CREATE_ Routine

To execute an SQL program from a COBOL program on a TNS/R system, use the CLU_PROCESS_CREATE_ routine. Use this routine for programs that run in the CRE or in a COBOL run-time environment on a TNS/R system. The CLU_PROCESS_CREATE_ routine calls the system procedure, PROCESS_CREATE_, and starts a new high-pin process by using the parameters that you supply. For more information, see the *CRE Programmer’s Guide*.

Running a Program in the OSS Environment

After successfully compiling your COBOL program with the `cobol` or the `nmcobol` utility in the OSS environment, you have an executable object file. Its name is either `a.out` (by default) or the name you gave it with the `-o` flag. If the current directory is in your search path, you can run your program by typing the name of the executable object file and pressing the Return key. For example, if the name of the executable object file is `a.out`, enter:

```
a.out
```

If the current directory is not in your search path, add it with this command:

```
export PATH=$PATH:.
```

In an OSS environment, you execute a program file by entering its name at the OSS shell prompt and pressing the Return key. The current directory must be in your search path. If the current directory is not in your search path, add it with this command:

```
export PATH=$PATH:.
```

You can also use the OSS `run` command to run a program file by using HP attributes (for example, a CPU or priority for the process). For example, to run the program with the Inspect symbolic debugger, enter:

```
run -debug a.out
```

For information about the `run` command, see the *Open System Services Shell and Utilities Reference Manual*.

The *COBOL85 for NonStop Systems Manual* also has detailed information on running COBOL programs from the OSS environment.

Running a Program at a Low PIN on a D-Series or Later Node

The operating system identifies a process (a running program) by a unique process identification number (PIN). In displays and printouts, a PIN usually appears after the number of the processor (CPU) where the process is running. For example, the operating system identifies a process in processor 4 with PIN 195 as 4,195.

The D-series operating system supports an architectural limit of 65,535 concurrent processes per processor. The actual number of concurrent processes depends on the available system resources (such as virtual memory) and the values specified during system generation.

A D-series or later PIN has these divisions:

- A low PIN ranges from 0 through 254.
- A high PIN ranges from 256 through 65,535 (or the maximum number).
- PIN 255 is reserved.

The information about D-series nodes also applies to G-series nodes. If an SQL program was written (or converted) to run at a high PIN, you usually want the program to run at a high PIN because more high PINs are available, and it frees the low PINs for processes that cannot run at a high PIN. If necessary, you can force the program to run at a low PIN interactively from a TACL process or programmatically from an application process. In a Pathway environment, you can also force a server process to run at a low PIN.

Interactive Commands

To interactively force an SQL program to run at a low PIN on a D-series node, use either of these methods:

- Before you run the SQL program, set the HIGHPIN object-file attribute to OFF in the SQL program file by using the Binder CHANGE command:

  ```
  @CHANGE HIGHPIN OFF IN sqlprog
  ```

  To change an object-file attribute in a program file, you must have read and write access to the program file. For a description of the Binder CHANGE command, see the *Binder Manual*.

- If you have not set the HIGHPIN object-file attribute to OFF (or cannot set it because of the file security), specify the HIGHPIN OFF run option in the TACL RUN command:

  ```
  RUN sqlprog / HIGHPIN OFF, ... /
  ```

Programmatic Commands

If a COBOL creator program must create an SQL program programmatically at a low PIN on a D-series node, consider these situations:

- The COBOL creator program was not written (or converted) to use the Common Run-Time Environment (CRE), or the COBOL creator uses the CRE but was compiled with the ENV OLD compiler directive.

  The SQL program runs at a low PIN by default, even if it was written (or converted) to run at a high PIN.

- The COBOL creator program was written (or converted) to run at a high PIN and to create a high PIN process. The SQL program was also written (or converted) to run at a high PIN.

  A high-PIN COBOL creator program that uses the CRE automatically creates a new process at a high PIN. Therefore, for the COBOL program to create the SQL program at a low PIN, you must set the HIGHPIN object-file attribute to OFF in the SQL program file by using the Binder CHANGE command as previously described under Interactive Commands on page 7-6.

Pathway Environment

In a Pathway environment, an SQL program running as a server process can run at an available high PIN if these conditions are met:

- The SQL program was written (or converted) to run at a high PIN.
- The HIGHPIN server attribute for the SQL program in the Pathway configuration file is ON.
- The HIGHPIN object-file attribute in the SQL program file is ON.
- A high PIN is available when the server runs.

If a Pathway server must run at a low PIN, use either of these methods:

- In the SQL program file, set the HIGHPIN object-file attribute to OFF by using the Binder CHANGE command as previously described under “Interactive Commands.”
- In the Pathway configuration file, set the HIGHPIN server attribute to OFF by using the SET SERVER or ALTER SERVER command. (The default for the HIGHPIN server attribute is OFF.)

Determining Compatibility With the SQL Executor

The PFV of an SQL program indicates the oldest version of the SQL executor that can execute the program. During SQL compilation, the SQL compiler writes the PFV in the program’s file label. Then, at run time, the SQL executor checks the PFV, and if the executor version is the same as or later than the program’s PFV, it executes the program. Otherwise, the executor returns an error.

To determine the version of the SQL executor, use the GET VERSION OF SYSTEM statement (all NonStop SQL/MP components on a system, including the executor, must have the same version). You can execute the GET VERSION OF SYSTEM statement from SQLCI or embedded in a COBOL program.

For a static embedded GET VERSION OF SYSTEM statement, include the INTO clause with a host variable:

```
EXEC SQL
  GET VERSION OF SYSTEM \NEWYORK INTO :HV-SYSTEM-VERSION
END-EXEC.
```

In this example, the statement returns the SQL/MP software version on the \NEWYORK system to the HV-SYSTEM-VERSION host variable. If you do not specify a system name, the statement returns the version of the local system.

To determine the PFV of an SQL program, use a FUP INFO or SQLCI FILEINFO command with the DETAIL option. For programs registered in version 300 (or later) catalogs, you can also query the PROGRAMS.PROGRAMFORMATVERSION column.

However, for version 300 (or later) SQL/MP software, HP recommends that you use the GET VERSION OF PROGRAM statement with the FORMAT option. You can enter this statement from SQLCI or embedded in a COBOL program. To embed a static GET VERSION OF PROGRAM statement in a COBOL program, you must include the INTO clause with a host variable. This statement returns the PFV of SQLPROG to the host variable named HV-PFV:

```
EXEC SQL
  GET FORMAT VERSION OF PROGRAM SQLPROG INTO :HV-PFV
END-EXEC.
```

You can also execute a dynamic GET VERSION OF PROGRAM statement by using the PREPARE and EXECUTE statements as shown in this example:

```
MOVE "GET FORMAT VERSION OF PROGRAM SQLPROG" TO HV-TEXT.
EXEC SQL
  PREPARE DYNAMIC-STATEMENT FROM :HV-TEXT
END-EXEC.
EXEC SQL
  EXECUTE DYNAMIC-STATEMENT RETURNING :HV-PFV
END-EXEC.
```

You cannot, however, use a GET VERSION OF PROGRAM statement with the EXECUTE IMMEDIATE statement.

For the complete syntax of the GET VERSION statements, see the *SQL/MP Reference Manual*.

8 Program Invalidation and Automatic SQL Recompilation

This section describes the causes of program invalidation and automatic SQL recompilation and preventive steps you can take in either case.

Program Invalidation

An SQL program file can be valid or invalid. A valid program can run without SQL recompilation using its current execution plans. An invalid program is subject to SQL recompilation (depending on options such as the similarity check) because of changes either to the program file itself or to an SQL object it references. An SQL program file has these classifications of SQL validity:

- The SQL SENSITIVE flag in the program's file label indicates the file is an SQL program that has been SQL compiled (although it might be invalid). The SENSITIVE flag also protects the program file from access by Enscribe utilities.
- The VALID flag in the program's file label and in the PROGRAMS catalog table indicates whether the program file can run without SQL recompilation.

SQL Compiler Validation Functions

The SQL compiler validates an SQL program file after a successful explicit SQL compilation or after errors occurred during a compilation with the FORCE option specified. During explicit compilation, the SQL compiler performs these functions related to program validation:

- Sets the VALID and SENSITIVE flags in the program's file label
- Records the timestamp of the SQL compilation in the program's file label
- Registers the program and sets the VALID flag in the PROGRAMS table
- Creates entries in the USAGES table for any SQL objects (tables, views, indexes, or collations) required by the program's execution plans

For a list of all SQL compiler functions, see Section 6, Explicit Program Compilation.

To determine if an SQL program is valid, use the SQLCI VERIFY utility or the SQLCI (or FUP) FILEINFO command with the DETAIL option. From a program, call the FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ system procedure and specify item codes 82 and 83. Item code 82 indicates whether the file is an SQL program (1=SQL program, 0=other), and item code 83 indicates whether the program file is valid (1=valid, 0=invalid).

Causes of Program Invalidation

Program invalidation is caused by certain operations performed on the program file and by DDL operations that alter an SQL object that the program references. During program invalidation, the SQL catalog manager performs these operations:

- Sets the VALID flag to N in the PROGRAMS catalog table and in the program's file label if the program file is accessible
- Deletes the program's usages entries in the USAGES table

An invalid SQL program must be recompiled either explicitly or automatically to generate valid execution plans before it can execute.

Operations Performed on an SQL Program File

These operations performed on an SQL program file cause the program file to be invalidated:

- Copying a program file. If you copy a program file by using the FUP or SQLCI DUP command, the original file is unaffected, but the new file is invalid.
- Binding or linking a program file. If you explicitly bind a program file by using the Binder program or link a program file by using the `nld` or `ld` utility, the original file is unaffected, but the resulting target file is invalid.
- Restoring a program file. If you restore a program file by using the RESTORE program without specifying the SQLCOMPILE ON option, the restored program becomes invalid.
- Running the Accelerator on a program file. If you run the Accelerator to optimize the object code, the program file becomes invalid.

Changes to Referenced SQL Objects

These changes to an SQL object cause a program file that references the object to be invalidated, except as described in Preventing Program Invalidation on page 8-4:

- Adding an index to a table, including an underlying table of a protection or shorthand view, by using the CREATE INDEX statement without the NO INVALIDATE option
- Adding a constraint, column, or partition on a table, including an underlying table of a protection or shorthand view
- Dropping a table or view
- Dropping a partition on a table or index
- Dropping an index or constraint on a table
- Moving a partition on a table
- Enabling or disabling the similarity check for a table or protection view

- Changing a collation, which includes dropping and then re-creating the collation, renaming a collation, or changing a DEFINE that points to a collation
- Executing an UPDATE STATISTICS statement with the RECOMPILE option for a table (RECOMPILE is the default option)
- Restoring a table, including an underlying table of a protection or shorthand view, by using the RESTORE program with the SQLCOMPILE OFF option specified

Changes to the AUDIT Attribute

Changing the AUDIT attribute of a table referenced by an SQL statement does not invalidate the program file. However, in these cases, changing the AUDIT attribute can cause automatic SQL recompilation (if it is allowed):

- If a statement performs a DELETE or UPDATE set operation on a nonaudited table with a SYNCDEPTH of 1, the SQL executor returns SQL error 8203 and forces the automatic recompilation of the statement.
- If a statement is executed in parallel on a table whose AUDIT attribute has changed since the last explicit SQL compilation, the SQL executor returns SQL error 8207 and forces the automatic recompilation of the statement.

Operations That Do Not Invalidate a Program File

These operations performed on an SQL program file or to an SQL object referenced by an SQL program file do not invalidate the program file:

- Renaming a program file
- Altering the security or owner of a program file or an SQL object
- Restoring a program file by using the RESTORE program with the SQLCOMPILE ON option specified
- Creating a view on a table
- Altering the file attributes of a table, except for changes to the AUDIT attribute as described in Changes to the AUDIT Attribute on page 8-3
- Adding an index to a table by using the CREATE INDEX statement with the NO INVALIDATE option
- Adding or dropping comments on an SQL object
- Executing an UPDATE STATISTICS statement with the NO RECOMPILE option specified for a table

File-Label and Catalog Inconsistencies

Because NonStop SQL/MP records SQL validity in both the program's file label and in the PROGRAMS catalog table, inconsistencies can occur. An invalid program file is sometimes recorded as valid in the catalog, or a valid program file is recorded as invalid in the catalog. Consider these situations:

- A program file is not accessible to the SQL catalog manager.

  A DDL operation alters an SQL object referenced by a program file. The SQL catalog manager marks the program as invalid in the PROGRAMS table, but then finds that the file is not accessible. The invalid program file remains marked as valid in its file label. At run time, however, the SQL executor performs the timestamp check for the referenced SQL object. When the timestamp check fails, the SQL executor invokes the automatic recompilation of the program.

- An SQL compiler (SQLCOMP) process abends.

  An event such as a CPU failure causes an SQLCOMP process to abend after it has generated a program file, marked the program file label as valid, and registered the program in the PROGRAMS table. TMF backs out the changes to the PROGRAMS table but not to the program's file label, because the file label is not audited. Therefore, a seemingly valid SQL program exists on disk, but an entry for the program does not exist in the catalog.

  You can sometimes recover from this condition by running SQLCOMP again to reenter the information in the catalog. However, you might first need to use the CLEANUP or GOAWAY utility to remove the invalid program file.

- The SQL catalog manager (SQLCAT) process abends.

  A DDL operation (described in Changes to Referenced SQL Objects on page 8-2) causes a program file to be marked as invalid both in the PROGRAMS table and in the program's file label. Then, an event such as a CPU failure causes the SQLCAT process to abend. TMF backs out the changes to the PROGRAMS table but not to the program's file label, because the file label is not audited. The valid SQL program file remains marked as invalid. To recover, you must re-execute the original DDL operation.

Preventing Program Invalidation

Compiling a program with the CHECK INOPERABLE PLANS option can prevent certain DDL operations from invalidating the program file. These DDL operations do not invalidate a program compiled with the CHECK INOPERABLE PLANS option if the similarity check is also enabled for each referenced object:

- ALTER TABLE...ADD PARTITION statement
- ALTER TABLE...ADD COLUMN statement (for more information, including restrictions, see ALTER TABLE... ADD COLUMN Statement and the Similarity Check on page 8-13)

- ALTER TABLE statement to move or split partitions (including a simple move, one-way split, or two-way split)
- ALTER TABLE...DROP PARTITION statement
- ALTER INDEX...DROP PARTITION statement (if the similarity check is enabled for the base table)
- ALTER INDEX statement to move or split index partitions
- CREATE INDEX statement
- UPDATE STATISTICS...RECOMPILE statement

The program also retains its entries in the USAGES table. These operations, however, do update the redefinition timestamp of each referenced object in the DDL statement.

The ALTER TABLE ... RENAME, ALTER INDEX ... RENAME, and ALTER INDEX ... ADD PARTITION statements do not invalidate a program regardless of whether it was compiled with the CHECK INOPERABLE PLANS option.

Note. These DDL operations always invalidate a program, even if the program was compiled with the CHECK INOPERABLE PLANS option:

- ADD CONSTRAINT statement
- DROP CONSTRAINT statement
- DROP TABLE statement
- DROP VIEW statement
- ALTER TABLE or ALTER VIEW statement with the SIMILARITY CHECK clause (for more information, see Enabling the Similarity Check for Tables and Protection Views on page 8-10)
- DROP INDEX statement, if the program contains a plan that references the dropped index

---

Automatic SQL Recompilation

Automatic SQL recompilation is the run-time SQL compilation, invoked by the SQL executor, of either an entire SQL program or a single static SQL statement in the program, depending on whether the RECOMPILE or RECOMPILEONDEMAND option was specified during explicit SQL compilation.

Automatic SQL recompilation validates only the copy of the SQL program or statement in memory. It does not validate the SQL program file on disk. Only explicit SQL compilation validates an SQL program file on disk.

Automatic SQL recompilation uses the default volume and catalog settings used for the explicit SQL compilation and the set of DEFINEs in effect at SQL load time (that is, when the SQL executor executes the first SQL statement in the program).

Automatic SQL recompilation performs these functions:

- Uses the current description of the database to determine the most efficient access path for each referenced database object
- Maximizes database availability and node autonomy by generating a new execution plan at run time
- Allows a program to reference database objects that did not exist during explicit SQL compilation
- Allows a program to use a new set of DEFINEs to specify a different database (for example, a development database rather than a production database)

You can enable or disable automatic SQL recompilation when you explicitly SQL compile a program. The RECOMPILE option (the default) enables automatic SQL recompilation, whereas the NORECOMPILE option disables it.

Causes of Automatic Recompilation

If automatic SQL recompilation is enabled (the NORECOMPILE option is not specified), the SQL executor invokes the SQL compiler to recompile a program or statement (depending on the RECOMPILEALL or RECOMPILEONDEMAND option) in these situations:

- The program file is marked invalid at SQL load time.
- The DEFINE values at SQL load time are different from the DEFINE values used to explicitly SQL compile the program.
- The timestamp check fails for an SQL object referenced in an SQL statement.
- An unavailable access path (index) exists.
- The program file contains an uncompiled SQL statement.

In some cases, you can prevent automatic recompilation by using the similarity check. For more information, see Preventing Automatic Recompilations on page 8-9.

Invalid SQL Program File

SQL load time occurs when the SQL executor executes the first SQL statement in a program. If the SQL program on disk is invalid for any of the reasons listed in Causes of Program Invalidation on page 8-2, the SQL executor forces the recompilation of the program or statement. To control the automatic recompilation, specify the RECOMPILEALL option (the default) to cause the recompilation of the entire program or the RECOMPILEONDEMAND option to limit the recompilation to statements actually executed.

Changed DEFINEs

If the values of the DEFINEs used in the program at SQL load time differ from the values of the DEFINEs used for explicit SQL compilation, the SQL executor forces the automatic recompilation of the program or statement by using the new DEFINE values.

(For a dynamic SQL statement, the SQL compiler uses the current set of DEFINEs when the PREPARE or EXECUTE IMMEDIATE statement executes.)

Failed Timestamp Check

The SQL executor performs the timestamp check for each SQL object referenced in an SQL statement at table open time (the first time the table is opened). The timestamp check ensures that a statement's current execution plan uses a valid definition of each SQL object (table or view, or a dependent object such as an index or collation), even if the program file was not accessible when the invalidating operation was performed on the SQL object. (For operations that invalidate an SQL program, see Changes to Referenced SQL Objects on page 8-2.)

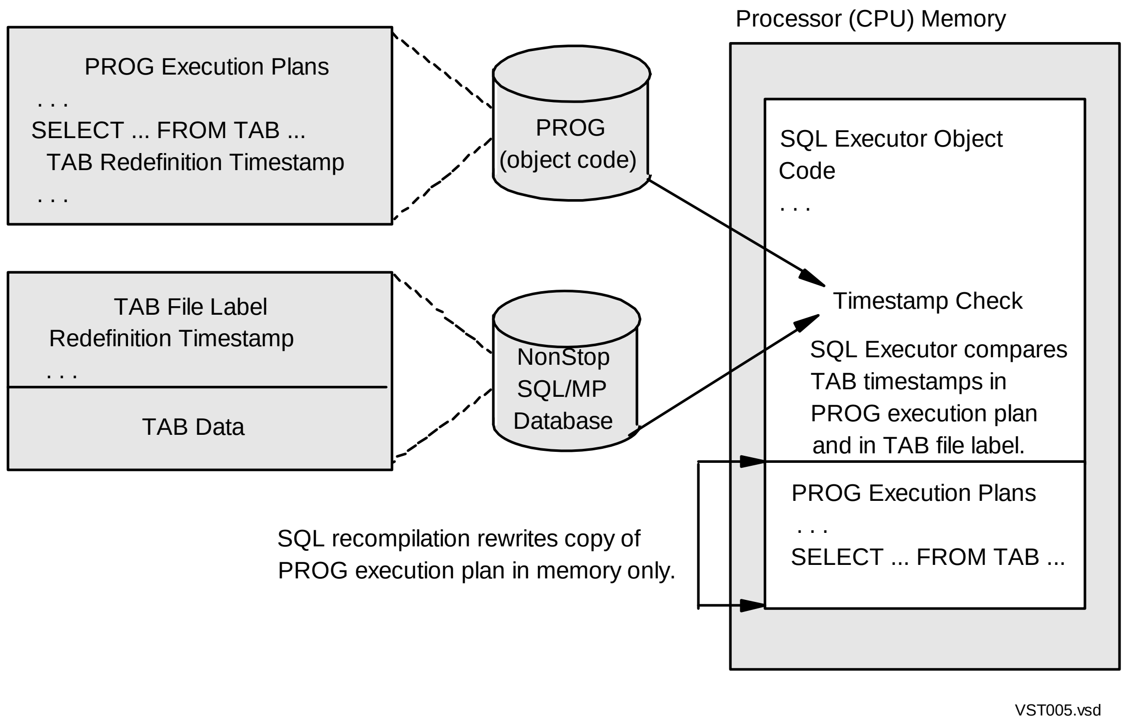
Each SQL object contains a redefinition timestamp in its file label. An SQL program file also contains the redefinition timestamps of all referenced SQL objects in each SQL statement's execution plan. When the SQL executor executes a statement, it compares the timestamp in the object's file label to the timestamp for the same object in the statement's execution plan. If the timestamps differ, the SQL executor forces a recompilation with the new definition of the object.

After opening a table, the SQL executor usually leaves a table open until the program stops running. However, a subsequent DDL or utility operation performed on the table (or a dependent object such as an index or collation), causes the table to be closed and its redefinition timestamp to be updated. If the SQL statement that refers to the table executes again, the SQL executor reopens the table and then performs the timestamp check, which forces a recompilation.

These steps describe the run-time timestamp check as shown in Figure 8-1 on page 8-8.

1. A valid SQL program named PROG refers to an SQL table named TAB in a SELECT statement. During explicit SQL compilation, SQL/MP generates an execution plan, which includes the TAB redefinition timestamp, for the SELECT statement and stores the plan in the PROG program file.
2. After PROG is running, a database administrator adds a new column to TAB using the ALTER TABLE statement. This operation updates the redefinition timestamp in the TAB file label.
3. When the SELECT statement executes, the SQL executor opens TAB and compares the timestamp in TAB file label with the TAB timestamp in the PROG execution plan. The TAB file label timestamp is more recent than the PROG execution plan timestamp. Therefore, the execution plan for the SELECT statement that was generated from the old definition of TAB during explicit SQL compilation is no longer valid.
4. The SQL executor invokes the SQL compiler to recompile the SELECT statement using the current TAB definition. This recompilation does not modify the PROG program file on disk; it only changes the copy of PROG in memory.

Figure 8-1. Timestamp Check



Unavailable Access Path (Index)

If the SQL executor encounters an unavailable access path (index) in the execution plan of an SQL statement, the SQL executor invokes the SQL compiler to recompile the statement. The SQL compiler then determines the best alternate access path, if such a path exists, to execute the statement. The SQL compiler recompiles only the affected SQL statement when an access path is unavailable.

Uncompiled SQL Statement

If the SQL executor encounters an uncompiled SQL statement, it invokes the SQL compiler to compile the statement. An SQL program file can contain an uncompiled SQL statement in these cases:

- The SQL statement referenced an SQL object that did not exist or was unavailable during explicit SQL compilation.
- The SQL statement referenced a DEFINE that did not exist during explicit SQL compilation.
- The program was explicitly compiled with the SQLCOMP FORCE option, and the SQL statement generated an error.

Run-Time Recompilation Errors

If an automatic SQL recompilation is successful, the SQL statement executes. However, if the recompilation fails, the SQL executor returns compilation errors or warnings:

- Recompilation of a single statement. The SQL executor returns error information to the SQLCODE variable and the SQLCA structure (if declared).
- Recompilation of an entire program. If an entire program is recompiled, an SQL statement that causes an error or warning remains uncompiled and the SQL executor suppresses the error or warning message. If the SQL executor subsequently executes the uncompiled statement, the SQL executor tries again to recompile the statement. If the statement still causes a compilation error or warning, the SQL executor returns error information to the SQLCODE variable and the SQLCA structure (if declared).

Preventing Automatic Recompilations

The SQL executor can perform the similarity check for SQL objects to determine if an invalid execution plan is operable or inoperable. An operable plan is semantically correct and can execute correctly without SQL recompilation (although the plan might not be optimal), whereas an inoperable plan must be recompiled to execute correctly.

By performing the similarity check, the SQL executor recompiles only SQL statements that have inoperable execution plans. It executes other SQL statements by using their existing plans. Executing the similarity check for an SQL statement eliminates unnecessary recompilations and is much faster than recompiling the statement.

This subsection describes the CHECK option and its effect on the SQL executor at run time. The COMPILE option directs the SQL compiler to perform similarity checks during explicit SQL compilation to explicitly recompile only statements with inoperable plans. For more information about the CHECK and COMPILE options, including their syntax, see Section 6, Explicit Program Compilation.

To direct the SQL executor to perform similarity checks for a program at run time, follow these steps:

1. Explicitly compile the program by using the CHECK INOPERABLE PLANS option.
2. Enable the similarity check by using DDL statements for each table or protection view referenced in the program. (SQL/MP implicitly enables the similarity check for other SQL objects.)

Note. You cannot use the similarity check for a query that uses parallel execution plans. At run time, a query that uses parallel execution plans will fail the similarity check, and the SQL statement containing the query must be automatically recompiled before it can execute (if NORECOMPILE is not specified). To use the similarity check in this query, you must disable parallel execution by using a CONTROL QUERY PARALLEL EXECUTION OFF directive.

Specifying the CHECK INOPERABLE PLANS Option

To direct the SQL executor to use the similarity check for a program, specify the CHECK INOPERABLE PLANS option when you explicitly compile the program as shown in the next example:

```
SQLCOMP /IN sqlprog,OUT $s.#sqlist/ CHECK INOPERABLE PLANS
```

For the complete syntax of the CHECK option, see Section 6, Explicit Program Compilation.

The CHECK INOPERABLE PLANS option directs the SQL compiler to store similarity information in the program file. The SIMILARITYINFO column in the PROGRAMS table indicates whether a program file contains similarity information:

Y	The execution plans in the program file contain similarity information.
N	The program file does not contain similarity information.

To use the CHECK INOPERABLE PLANS option, you must have an SQL/MP software version of 310 or later. If you specify a CHECK option, the SQL compiler sets the program's PFV to 310 (or later). The SQL compiler also sets the program's PCV to 310 (or later). Therefore, the SQL catalog in which the program is registered must have a catalog version of 310 (or later).

For more information, see the *SQL/MP Version Management Guide.*

Enabling the Similarity Check for Tables and Protection Views

To use the CHECK INOPERABLE PLANS option, the similarity check must be enabled for any referenced tables or protection views at run time. You must explicitly enable the similarity check for a table or protection view, including any underlying tables for the view, as shown in these DDL statements. (SQL/MP implicitly enables the similarity check for other SQL objects.)

```
CREATE TABLE table-name ...
                  [ SIMILARITY CHECK { ENABLE | DISABLE } ]

CREATE VIEW view-name ...
                    FOR PROTECTION
                      ...
                  [ SIMILARITY CHECK { ENABLE | DISABLE } ]

ALTER TABLE table-name ...
                  [ SIMILARITY CHECK { ENABLE | DISABLE } ]

ALTER VIEW view-name ...
                  [ SIMILARITY CHECK { ENABLE | DISABLE } ]
```

`table-name` or `view-name`

is the Guardian name or DEFINE name of the table or protection view. The name cannot be a shorthand view. For the ALTER TABLE statement with the SIMILARITY CHECK clause, `table-name` cannot be an SQL catalog table.

`SIMILARITY CHECK ENABLE | DISABLE`

enables or disables the similarity check for the specified table or protection view. DISABLE is the default.

For the complete syntax of these statements, see the *SQL/MP Reference Manual*.

If you use the ALTER TABLE or ALTER VIEW statement to change the similarity check attribute, the SQL catalog manager invalidates any programs, as identified in the USAGES table, that reference the table or protection view. If the ALTER TABLE or ALTER VIEW statement sets the similarity check attribute to its current value, programs are not invalidated.

If you enable the similarity check for a protection view, the operation does not enable the check for any underlying tables. You must explicitly enable the similarity check for the underlying table. If you enable the similarity check for an underlying table, the operation does not enable the check for a protection view defined on the table.

The SIMILARITYCHECK column in the TABLES table indicates whether a table or protection view has the similarity check enabled:

ENABLED	The similarity check is enabled.
DISABLED	The similarity check is disabled.

A table or protection view that has the similarity check enabled has a version of 310 (or later). All SQL/MP components, including the executor, catalog manager, and compiler, must have a software version of 310 (or later) to access the table or protection view. An SQL catalog that supports the similarity check must have a catalog version of 310 (or later). For more information, see the *SQL/MP Version Management Guide.*

Similarity Rules for Tables

For two tables to be similar, the characteristics and attributes of the tables must be the same, except for the differences listed in this subsection. These tables are used to describe these differences:

- COMPILE-TIME-TABLE is the table SQLCOMP uses to generate the execution plan during explicit SQL compilation. COMPILE-TIME-TABLE must have the similarity check enabled for the COMPILE INOPERABLE PLANS option. (If the similarity check is not enabled for COMPILE-TIME-TABLE, the CHECK INOPERABLE PLANS option returns SQL warning 4315.)
- RUN-TIME-TABLE is the table the program accesses at run time. RUN-TIME-TABLE must have the similarity check enabled for the CHECK

INOPERABLE PLANS option. Otherwise, the similarity check fails and automatic recompilation occurs.

RUN-TIME-TABLE can be the same table as COMPILE-TIME-TABLE, a modified version of COMPILE-TIME-TABLE, or a different table altogether.

Note. The similarity check does not support parallel execution plans. Tables are not considered similar if they are specified in a query that uses a parallel execution plan.

For RUN-TIME-TABLE to be similar to COMPILE-TIME-TABLE, all characteristics and attributes must be the same, except for these allowable differences:

- Names of the tables.
- Contents of the tables (that is, the data in the table).
- Partitioning attributes (number of partitions and partitioning key ranges).
- Number of indexes–RUN-TIME-TABLE must have all indexes used by COMPILE-TIME-TABLE in the execution plan. RUN-TIME-TABLE can also have additional indexes that COMPILE-TIME-TABLE does not have. COMPILE-TIME-TABLE can have indexes that RUN-TIME-TABLE does not have but only if the execution plan does not use the additional indexes.
- Key tags (or values) for indexes.
- Creation timestamp and redefinition timestamp.
- AUDIT attribute–If, however, a statement performs a DELETE or UPDATE set operation on a nonaudited table that has a SYNCDEPTH of 1, the SQL executor returns an error and forces the automatic recompilation of the statement (if NORECOMPILE is not specified).
- Any of these file attributes:

ALLOCATE	LOCKLENGTH	SECURE
AUDITCOMPRESS	MAXEXTENTS	SERIALWRITES
BUFFERED	NOPURGEUNTIL	TABLECODE
CLEARONPURGE	OWNER	VERIFIEDWRITES
EXTENT (primary and secondary		

- Statistics on the tables.
- Column headings.
- Comments on columns, constraints, indexes, or tables.
- Catalog where the table is registered.
- Help text.
- Number of columns–RUN-TIME-TABLE can have more columns than COMPILE-TIME-TABLE, but the common columns of both tables must have identical attributes. However, if a statement uses a SELECT list containing an

asterisk (*), RUN-TIME-TABLE must have the same number of columns as COMPILE-TIME-TABLE. For more information, see the next subsection.

Similarity Rules for Protection Views

The similarity check does not support shorthand views. The similarity rules for protection views are:

- A protection view is never similar to a table or other SQL object.
- To pass the similarity check, two protection views must follow this criteria:
  - Have similar underlying base tables
  - Project the same columns from the base tables
  - Have the same column names
  - Have the same selection expression, which is determined by a binary comparison of the generated objects for the two selection expressions

ALTER TABLE... ADD COLUMN Statement and the Similarity Check

Two tables are not required to have the same number of columns to pass the similarity check, but tables with a different number of columns must observe these restrictions (as well as the other similarity check rules) to pass the check:

- The number of columns in COMPILE-TIME-TABLE must be less than or equal to the number of columns in RUN-TIME-TABLE.
- The common columns of the tables must have identical attributes. For example, if COMPILE-TIME-TABLE has five columns, RUN-TIME-TABLE can have more than five columns, but the first five columns of each table must be identical.

Therefore, you can use the ALTER TABLE ... ADD COLUMN statement for a table without forcing the recompilation of a program that accesses the table. However, these cases show several problems that can occur when you use the ALTER TABLE ... ADD COLUMN statement and the similarity check.

An SQL statement uses an asterisk (*) in a select list with the similarity check for tables with a different number of columns as shown in these statements:

Statement	Similarity Check Results
`SELECT * FROM` *`table1`*	TABLE1 = Fail
`SELECT DISTINCT * FROM` *`table1`*	TABLE1 = Fail
`SELECT COUNT (*) FROM` *`table1`*	TABLE1 = Pass
`SELECT` *`columna`* `FROM` *`table1`* `WHERE` *`columnb`* `relation-operator (SELECT COUNT(*) FROM` *`table2`*`)`	TABLE1 = Pass, TABLE2 = Pass

`SELECT columna FROM table1`
`  WHERE EXISTS`
`    (SELECT [DISTINCT] * FROM table2)`
	TABLE1 = Pass
	TABLE2 = Fail

`INSERT INTO table1`
`  (SELECT [DISTINCT] * FROM table2)`
	TABLE1 = Fail
	TABLE2 = Fail

`SELECT table1.*,table2.x`
`  FROM table1,table2`
	TABLE1 = Fail,
	TABLE2 = Pass

An SQL statement uses unqualified column names and the additional columns make one of the column names used in the statement ambiguous. When the statement is compiled, the column names are resolved unambiguously. However, if the execution plan for the statement is executed against a RUN-TIME-TABLE with more columns than the COMPILE-TIME-TABLE, the column names might not be resolved unambiguously.

For example, consider these SQLCI commands:

```
CREATE TABLE table1 (a INTEGER, b INTEGER);
INSERT INTO table1 VALUES (11,22);

CREATE TABLE table2 (c INTEGER, d INTEGER);
INSERT INTO table2 VALUES (33,44);

PREPARE statement1 FROM SELECT a,b,c,d FROM table1, table2;
EXECUTE statement1; -- Returns 11,22,33,44

ALTER TABLE table1 ADD COLUMN c INTEGER DEFAULT NULL;
PREPARE statement1; -- Returns an error because the compiler
                    -- cannot resolve column c unambiguously
```

A similar situation occurs if you specify the CHECK INOPERABLE PLANS option and execution-time name resolution. When the SQL executor attempts to use the plan with a new set of tables, it retains the association of the unqualified column names with tables established when the statement was explicitly compiled. However, if the similarity check fails and automatic recompilation is attempted, the recompilation also fails because of the ambiguity.

If an INSERT statement does not specify the column-name list, the statement must specify values for all the columns in the table:

```
INSERT INTO table1 VALUES (1,2,3,4);
INSERT INTO table1 (SELECT a,b,c,d FROM table2);
```

For these statements to compile successfully, TABLE1 must have four columns at both compile time and run time. A program cannot use the CHECK INOPERABLE PLANS option to execute the statement against TABLE1 after a column has been added to the run-time version of TABLE1. In this case, the similarity check fails, and the statement is automatically recompiled.

Collations

You do not have to explicitly enable the similarity check for a collation, because collations always have the check implicitly enabled. Two collations are similar only if they are equal. SQL/MP uses the CPRL_COMPAREOBJECTS_ procedure to compare the two collations. Consequently, two tables that contain character columns associated with collations are similar only if the collations are equal.

9 Error and Status Reporting

This section provides information about error and status reporting after the execution of an SQL statement or directive in a COBOL program. For information about the SQL descriptor area (SQLDA), see Section 10, Dynamic SQL Operations.

Topics include:

- Using the INCLUDE STRUCTURES Directive
- Returning Error and Warning Information on page 9-4
- Returning Performance and Statistics Information on page 9-21

Using the INCLUDE STRUCTURES Directive

The INCLUDE STRUCTURES directive specifies the version of SQL structures that the HP COBOL compiler generates. You must specify the INCLUDE STRUCTURES directive to generate version 300 or later SQL data structures. If you omit this directive, the HP COBOL compiler generates version 2 structures by default and includes this informational message in the compilation summary:

```
INCLUDE STRUCTURES directive for SQL is missing.  SQL
VERSION 2 is assumed.  This may produce incorrect SQL
results in programs which use features introduced
in SQL versions greater than VERSION 2.
```

Code the INCLUDE STRUCTURES directive in the declarations area of the procedure before you code an INCLUDE SQLCA, INCLUDE SQLSA, or INCLUDE SQLDA directive. If the procedure is part of a compilation unit that consists of more than one procedure, place the INCLUDE STRUCTURES directive in the global declarations area or in the declarations area of the first procedure. The directive then applies to all procedures in the compilation unit.

Use this syntax for the INCLUDE STRUCTURES directive:

```
INCLUDE STRUCTURES { structure-spec }

  structure-spec is:

      { [ ALL ] VERSION version                 }

      { { SQLCA | SQLSA | SQLDA } VERSION version }...

      { { SQLCA | SQLSA } [ EXTERNAL ]          }
```

`ALL VERSION`

are keywords that specify the same version for all three SQL structures (SQLCA, SQLSA, and SQLDA).

`{ SQLCA | SQLSA | SQLDA } VERSION`

are keywords that specify the SQLCA, SQLSA, or SQLDA structure, respectively.

`version`

is the version number of the generated data structures. *`version`* can be 1, 2, 300, or later.

`{ SQLCA | SQLSA } [ EXTERNAL ]`

specifies that the structures are declared as external, making it possible to share them among subprograms of the main program.

As a result, the SQLCA object or the SQLSA object can be referenced by any subprogram in the main program that describes the object. All such descriptions must be identical, or the results of the references are unpredictable. References to an external object from different programs are always to the same object. In the main program, there is only one representation of an external object. The storage associated with that object is associated with the main program rather than with any particular program within it.

Generating Structures With Different Versions

You can generate SQL structures that are all the same version or structures of different versions. For example, to generate all version 310 structures in a program, specify this directive:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 310 END-EXEC.
```

Or, to generate different versions for each structure, specify this directive:

```
EXEC SQL INCLUDE STRUCTURES
                 SQLCA VERSION 2
                 SQLSA VERSION 2
                 SQLDA VERSION 310
END-EXEC.
```

Checking the Version of the HP COBOL Compiler

If you try to compile a COBOL program that uses the INCLUDE STRUCTURES directive to specify a later version of a structure than the HP COBOL compiler can generate, the compiler returns SQL error 11203. To determine the version of the HP COBOL compiler before you compile a program, run the VPROC program for the HP COBOL compiler object file. Then, check the version in the VPROC line that contains S7094, which is the SQL compiler interface (SCI) product number.

When you compile the program, you can specify the SQLMAP option in the SQL compiler directive. The SQLMAP option directs the HP COBOL compiler to include the host object SQL version (HOSV) in the map at the end of the source-file listing. For example, a version 310 HP COBOL compiler listing includes this line:

```
Host Object SQL Version = 310
```

For more information about versions of NonStop SQL/MP, see the *SQL/MP Version Management Guide*.

Sharing Structures

Sharing a single SQLCA and SQLSA structure among subprograms of a host program saves a large amount of memory space. The SQLCA structure is 430 bytes. The pre-R330 SQLSA structure is 838 bytes, and the R330 SQLSA structure is 1790 bytes. A program with many subprograms that contain embedded SQL can consume enormous amounts of memory space for the multiple structures alone.

The COBOL external object generated by the INCLUDE SQLCA EXTERNAL directive is:

```
01  SQLCA EXTERNAL.
   02 EYE-CATCHER         PIC X(2).
   02 VERSION             PIC S9(4) COMP.
   02 NUM-ERR-ENTRIES     PIC S9(4) COMP.
   02 PARAMS-BUFFER-LEN   PIC S9(4) COMP.
   02 SRC-NAME-BUFFER-LEN PIC S9(4) COMP.
   02 NUM-ERRORS          PIC S9(4) COMP.
   02 NEXT-P-OFFSET       PIC S9(4) COMP.
   02 FLAGS               PIC S9(4) COMP.
   02 PROCEDURE-ID        PIC X(32).
   02 USER-LINE-NUMBER    PIC S9(9) COMP.
   02 SYNTAX-ERR-LOC      PIC S9(4) COMP.
   02 ERROR-LOCATION      PIC X(40).
   02 ROWS                PIC S9(9) COMP.
   02 COST                PIC S9(18) COMP.
   02 SQLCA-RESERVED      PIC X(40).
   02 ERRORS-ALL.
       03 SQL-ERROR        OCCURS 7 TIMES.
          04 ERRCODE          PIC S9(4) COMP.
          04 SUBSYSTEM-ID     PIC X.
          04 SUPPRESS-DISPLAY PIC X.
          04 PARAMS-OFFSET    PIC S9(4) COMP.
          04 PARAMS-COUNT     PIC S9(4) COMP.
          04 ARRIVAL-SEQ      PIC S9(4) COMP.
   02 SQLCODEA          REDEFINES ERRORS-ALL.
       03 SQLCODE          PIC S9(4) COMP.
       03 FILLER           PIC X(68).
   02 PARAMS-BUFFER        PIC X(180).
   02 SRC-NAME-BUFFER      PIC X(34).
```

The COBOL external object generated by the INCLUDE SQLSA EXTERNAL directive is:

```
01 SQLSA EXTERNAL.
   02 EYE-CATCHER         PIC X(2).
   02 VERSION             PIC S9(4) COMP.
   02 DML.

      03 NUM-TABLES          PIC 9(4) COMP.
      03 STATS OCCURS 16 TIMES.
```

```
            04 TABLE-NAME          PIC X(24).
            04 RECORDS-ACCESSED    PIC S9(9) COMP.
            04 RECORDS-USED        PIC S9(9) COMP.
            04 DISC-READS          PIC S9(9) COMP.
            04 MESSAGES            PIC S9(9) COMP.
            04 MESSAGE-BYTES       PIC S9(9) COMP.
            04 WAITS               PIC S9(4) COMP.
            04 ESCALATIONS         PIC S9(4) COMP.
            04 SQLSA-RESERVED      PIC X(4).
      02 PREPARE REDEFINES DML.
         03 INPUT-NUM           PIC 9(4) COMP.
         03 INPUT-NAMES-LEN     PIC 9(4) COMP.
         03 OUTPUT-NUM          PIC 9(4) COMP.
         03 OUTPUT-NAMES-LEN    PIC 9(4) COMP.
         03 NAME-MAP-LEN        PIC 9(4) COMP.
         03 SQL-STATEMENT-TYPE  PIC 9(4) COMP.
            88 SQL-STATEMENT-SELECT  VALUE 1.
            88 SQL-STATEMENT-INSERT  VALUE 2.
            88 SQL-STATEMENT-UPDATE  VALUE 3.
            88 SQL-STATEMENT-DELETE  VALUE 4.
            88 SQL-STATEMENT-DDL     VALUE 5.
            88 SQL-STATEMENT-CONTROL VALUE 6.
            88 SQL-STATEMENT-DCL     VALUE 7.
```

Returning Error and Warning Information

SQL/MP provides these methods you can use to check for and process errors and warnings in your program:

- Checking the SQLCODE (or SQLCODEX) data item
- Using the WHENEVER directive
- Using constraints to check for errors
- Checking information from the SQLCA structure

Checking the SQLCODE Identifier

SQL/MP returns an error or warning code to SQLCODE after the execution of each embedded SQL statement or directive:

Value	Status
< 0	Error
> 0	Warning
0	Successful

Each SQL/MP error or warning message has an assigned code. For the codes and their meanings, see the *SQL/MP Messages Manual*.

Using the SQLCODE Data Item

The HP COBOL compiler does not automatically generate an SQLCA structure, which includes the SQLCODE data item. You must declare an SQLCODE identifier either explicitly as a PIC S9(4) COMP data item or implicitly with an INCLUDE SQLCA directive. You cannot specify SQLCODE both explicitly as a data item and implicitly with an INCLUDE SQLCA directive.

This example uses the INCLUDE SQLCA directive to implicitly declare the SQLCODE data item:

```
 WORKING-STORAGE SECTION.
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 IN-PARTS-REC.
      10  IN-PARTNUM   PIC 9(4).
      10  IN-PRICE     PIC 9(6)V99.
      10  IN-PARTDESC  PIC X(18).
 EXEC SQL END DECLARE SECTION END-EXEC.

 EXEC SQL INCLUDE SQLCA END-EXEC.
...
```

This example checks SQLCODE only for errors and warnings:

```
 PROCEDURE DIVISION.
 ...
   300-INSERT-DATA.
       MOVE 4120 TO IN-PARTNUM.
       MOVE 60000.00 TO IN-PRICE.
       MOVE "V8 DISK OPTION" TO IN-PARTDESC.
       EXEC SQL
         INSERT INTO SALES.PARTS (PARTNUM, PRICE, PARTDESC)
           VALUES (:IN-PARTNUM, :IN-PRICE, :IN-PARTDESC)
       END-EXEC.
* Check for errors and warnings
       IF SQLCODE < 0 PERFORM 900-HANDLE-ERRORS.
       IF SQLCODE > 0 AND
          SQLCODE NOT = 100 PERFORM 910-HANDLE-WARNINGS.
  ...
```

This example checks for all values for SQLCODE:

```
 PROCEDURE DIVISION.
 ...
 IF SQLCODE OF SQLCA = 0
    PERFORM DISPLAY-ROW-3000
 ELSE
    IF SQLCODE OF SQLCA = 100
       PERFORM ROW-NOT-FOUND-7000
 ELSE
    IF SQLCODE OF SQLCA < 0
          PERFORM SQL-ERROR-9000
 ELSE
    IF SQLCODE OF SQLCA > 0
```

```
           PERFORM SQL-WARNING-8000
...
```

Using the SQLCODEX Data Item

You can use level-88 items with an SQLCODE data item, such as:

1. Substitute an SQLCODEX data item for the SQLCODE data item.

   To do this, include an INVOKE SQLCODEX statement in each program or nested program in which you want an SQLCODEX data item.

2. Attach level-88 items to the SQLCODEX data item.
3. Declare the SQLCA structure.

However, this approach is not recommended, because SQLCODEX is not part of the ANSI Database—Embedded SQL Standard. Rather, omit the INCLUDE SQLCA directive from each program or nested program to prevent the compiler from declaring the SQLCODE data item implicitly so that you can declare it explicitly and attach level-88 items to it.

Using the WHENEVER Directive

The WHENEVER directive specifies an action that a program takes, depending on the result of subsequent DML, DCL, and DDL statements. WHENEVER provides tests for these conditions:

- An error occurred.
- A warning occurred.
- No rows were found.

You must specify a WHENEVER directive in the Procedure Division after a section or paragraph name. When you specify this directive, the HP COBOL compiler inserts statements that perform run-time checking after an SQL statement using the SQLCODE variable. You do not have to explicitly check the SQLCODE value.

Although using WHENEVER directives is simpler than writing a routine to test SQLCODE, you still must code the error routines. Typically, you code the routines once and save them in a library file to be copied into your programs as needed. You can use the COBOL SOURCE directive to copy the error routines into your program.

This table indicates the HP COBOL compiler pseudocode that checks SQLCODE and the order in which the checks are made:

Order	Condition	Compiler Pseudocode
1	NOT FOUND	`IF SQLCODE = 100 THEN` *action-specification*
2	SQLERROR	`IF SQLCODE < 0 THEN` *action-specification*
3	SQLWARNING	`IF SQLCODE > 0 AND SQLCODE NOT = 100 THEN` *action-specification*

action-specification is one of:

```
PERFORM :host-identifier ;
GOTO :host-identifier ;
GO TO :host-identifier;
CONTINUE ;
```

When more than one WHENEVER condition applies to an SQL statement, SQL/MP processes the conditions in order of precedence. For example, an SQL error and an SQL warning can occur for the same statement, but the error condition has a higher precedence and is processed first.

These WHENEVER directives check for the error, warning, and not-found conditions:

```
EXEC SQL
  WHENEVER NOT FOUND PERFORM :ROW-NOT-FOUND-7000 END-EXEC.
EXEC SQL
  WHENEVER SQLERROR PERFORM :SQL-ERROR-9000 END-EXEC.
EXEC SQL
  WHENEVER SQLWARNING PERFORM :SQL-WARNING-8000 END-EXEC.
...
```

Note. SQL/MP sometimes returns values other than 100 for a not-found condition. For example, SQL error 8230 indicates that a subquery did not return any rows, and SQL error 8423 indicates that an indicator variable was not specified for a null output value.

Determining the Scope of a WHENEVER Directive

The order in which WHENEVER directives appear in the listing determines their scope. Some considerations follow:

- A WHENEVER directive remains in effect until another WHENEVER directive for the same condition appears. If you want to execute a different routine when an error occurs, specify a new WHENEVER directive with a different PERFORM routine.

  For example, to insert a new row only when a row is not found, specify a new WHENEVER directive:

  ```
  EXEC SQL WHENEVER NOT FOUND PERFORM :INSERT-ROW END-EXEC.
  ```

  The new WHENEVER directive remains in effect until it is disabled or changed.

- If another program is called within the error handling code, the position of the called program in the listing order determines the WHENEVER directive in effect. The context of the calling program has no effect.
- The listing order includes files copied into the program through a SOURCE directive. If a copied file contains a WHENEVER directive, that directive is in effect following the SOURCE directive.
- SQL statements are not affected by the WHENEVER directive if they appear in the program before the WHENEVER directive enables condition checking.
- Do not code WHENEVER directives inside IF statements. A COBOL terminator on a WHENEVER directive or any other nonexecutable statement, such as DECLARE CURSOR or CONTROL statement, does not affect execution.

Enabling and Disabling WHENEVER Checking

You can enable and disable the WHENEVER directive for different parts of your program. For example, you might want to handle SQL errors by checking SQLCODE after an SQL statement instead of using WHENEVER SQLERROR. Example 9-1 on page 9-9 shows how to enable and disable the WHENEVER directive.

Example 9-1. Enabling and Disabling the WHENEVER Directive

```
 PROCEDURE-DIVISION.

 0010-SET-UP.
      EXEC SQL
        WHENEVER SQLERROR PERFORM :9900-SQL-ERROR-HANDLER
      END-EXEC.
...
 9900-SQL-ERROR-HANDLER.
* Disable SQLERROR handling to prevent looping.
      EXEC SQL WHENEVER SQLERROR END-EXEC.
     ...
      EXEC SQL
        INSERT INTO ERRLOG
               ( ..., ERRORS_SQL, ...)
        VALUES ( ..., :SQLCODE-NUM, ...)
      END-EXEC.
     ...

       ...
      EXEC SQL
        INSERT INTO ERRLOG
               ( ..., ERRORS_SQL, ...)
        VALUES ( ..., :SQLCODE-NUM, ...)
      END-EXEC.
     ...
* Enable SQLERROR handling for subsequent statements.
      EXEC SQL
        WHENEVER SQLERROR PERFORM :9900-SQL-ERROR-HANDLER
      END-EXEC.

* Exit code
```

Avoiding Infinite Loops

To avoid an infinite loop if the error handling code generates errors or warning, you can disable the WHENEVER directive within the error handling procedure. An infinite loop can occur in these situations:

- The SQLERROR condition executes a statement that generates an error.
- The SQLWARNING condition executes a statement that generates a warning.
- The NOT FOUND condition executes a statement that generates a NOT FOUND condition.

To avoid these situations, disable the appropriate WHENEVER directive for the part of your program that handles the condition. Example 9-2 on page 9-11 enables and disables the WHENEVER directive.

Using an Aggregate Function

All aggregate functions except COUNT return a null value when operating on an empty set. If a host variable receives the null value as the result of an aggregate function, you must specify an indicator variable and test the result of the indicator variable. Otherwise, SQL/MP returns a “no indicator variable provided” condition instead of a “no rows found” condition. A WHENEVER NOT FOUND directive does not detect this condition.

Example 9-2 on page 9-11 illustrates the use of WHENEVER directives that detect error and warning conditions. For the INSERT statement, the SQLERROR directive is processed first. This directive has a higher precedence, although the SQLWARNING directive is specified first in the source code.

Example 9-2. Using the WHENEVER Directive

```
WORKING-STORAGE SECTION.
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 IN-PARTS-REC.
      10  IN-PARTNUM   PIC 9(4).
      10  IN-PRICE     PIC 9(8)V99.
      10  IN-PARTDESC  PIC X(18).
 EXEC SQL END DECLARE SECTION END-EXEC.

 01 WARNING-SUM   PIC S9(4) COMP.

 EXEC SQL INCLUDE SQLCA END-EXEC.
 ...
 PROCEDURE DIVISION.
 100-OPENING.
 EXEC SQL WHENEVER SQLWARNING PERFORM :910-WARNINGS END-EXEC.
 EXEC SQL WHENEVER SQLERROR PERFORM   :900-ERRORS   END-EXEC.
 ...
 300-WORK.
     PERFORM 5000-BEGIN-TRANSACTION.
*    Execute an SQL INSERT into the PARTS table.
     MOVE 4120 TO IN-PARTNUM.
     MOVE 60000.00 TO IN-PRICE.
     MOVE "V8 DISK OPTION" TO IN-PARTDESC.
 EXEC SQL INSERT INTO SALES.PARTS (PARTNUM, PRICE, PARTDESC)
     VALUES (:IN-PARTNUM, :IN-PRICE, :IN-PARTDESC)
 END-EXEC.

 EXEC SQL DELETE FROM SALES.PARTS WHERE QTY_AVAILABLE = 0
 END-EXEC.

     PERFORM 6000-COMMIT-TRANSACTION.
     STOP RUN.
 900-ERRORS.
     DISPLAY "SQLCODE = " SQLCODE.
     EXEC SQL ROLLBACK WORK END-EXEC.
     DISPLAY " TRANSACTION ABORTED. "
     STOP RUN.
 910-WARNINGS.
     ADD 1 TO WARNING-SUM.
     DISPLAY " WARNING: SQLCODE = " SQLCODE.

 5000-BEGIN-TRANSACTION.
      EXEC SQL BEGIN WORK END-EXEC.

 6000-COMMIT-TRANSACTION.
      EXEC SQL COMMIT WORK END-EXEC.
      DISPLAY " COMMIT TRANSACTION".
```

Returning Information From the SQLCA

SQL/MP returns run-time information, including errors and warnings, for the most recently executed SQL statement to the SQL communication area (SQLCA). The SQLCA structure can contain up to seven error or warning codes (in any combination) that might be returned by a single SQL statement or directive.

Declaring the SQLCA Structure

The HP COBOL compiler does not automatically generate an SQLCA structure. You must explicitly declare an SQLCA structure using an INCLUDE SQLCA directive in the Data Division of your program. To declare an SQLCA structure, specify the INCLUDE SQLCA directive using this syntax (if you do not first specify the INCLUDE STRUCTURES directive, SQL/MP generates version 2 structures by default):

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

Using System Procedures With the SQLCA Structure

The SQL system procedures are written in TAL, but you can call them from a COBOL program using an ENTER TAL statement. Use the SQL system procedures to return SQLCA information:

System Procedure	**Description**
SQLCA_DISPLAY2_	Writes SQL error and warning messages from the SQLCA structure to a file or terminal
SQLCAGETINFOLIST	Writes a specified subset of the SQL error or warning information from the SQLCA structure to a record area in the program
SQLCA_TOBUFFER2_	Writes SQL error or warning messages from the SQLCA structure to a record area in the program
SQLCAFSCODE	Returns information about file-system, disk-process, or operating system errors returned to the program

For more information, see Section 5, SQL/MP System Procedures.

Table 9-1 on page 9-13 describes the SQLCA fields generated by the INCLUDE SQLCA directive. Do not access the SQLCA fields directly. HP reserves the right to change the SQLCA fields in future PVUs.

Table 9-1. SQLCA Structure Fields (page 1 of 3)

Field Name	Description
EYE-CATCHER	Identification field, always set by the system to CA.
VERSION-ID	Current version of the SQLCA; subsequent SQL/MP software PVUs can change this value.
NUM-ERR-ENTRIES	Maximum number of error entries that the ERRORS-ALL group item can hold. The number is 7.
PARAMS-BUFFER-LEN	Byte length of the PARAMS-BUFFER item. The maximum length is 180 bytes.
SRC-NAME-BUFFER-LEN	Length of the buffer that contains the name of the program source file.
NUM-ERRORS	Current number of errors or warnings returned to the ERRORS-ALL group item.
NEXT-P-OFFSET	First empty byte in PARAMS-BUFFER. By subtracting this number from 180, you can determine the remaining space in the buffer. The first byte is byte 0.
FLAGS	Code for a flag. Interpret the values: 0 No flags 1 More errors or warnings than ERRORS-ALL can hold 2 More parameters than PARAMS-BUFFER can hold 3 Both 1 and 2 are true
PROCEDURE-ID	Program ID of the program that contains the SQL statement receiving the error or warning messages. If there are no error or warning messages, this field is blank.
USER-LINE-NUMBER	Source code line number of the SQL statement that causes errors or warnings (zero if none occurred).
SYNTAX-ERR-LOC	Character position in the SQL statement where the syntax error occurs. The value -1 indicates that the error is not a syntax error.
ERROR-LOCATION	Buffer that contains the name of the system procedure that detected the error and the offset within the procedure where the error occurred.
ROWS	Number of rows updated, deleted, or inserted in the table or protection view. If there is an error on the statement, the value of ROWS is an approximate count. It could be less than the number actually changed. This item is set to 0 for any statement except UPDATE, DELETE, and INSERT.
COST	Execution cost of a query as estimated by the SQL compiler (set only for dynamic SQL programs).
SQLCA-RESERVED	Reserved
ERRORS-ALL	Group item for error information.

Table 9-1. SQLCA Structure Fields (page 2 of 3)

Field Name	Description
SQL-ERROR	Repeating group for error information. There are 7 occurrences, each of which returns information for a single error or warning.
ERRCODE	Error and warning messages are documented in the *SQL/MP Messages Manual.*
SUBSYSTEM-ID	One-byte code that identifies the system component issuing the error or warning: D DP2 disk process F SQL file system G NonStop OS I Sequential I/O (SIO) procedures L Load routines S SQL/MP component: (SQL compiler, catalog manager, executor, SQLUTIL, SQLCI, SQLCI2) R SORTPROG process (FastSort program)
SUPPRESS-DISPLAY	Code that indicates whether to suppress the display. The system component issuing the message sets this data item to Y (for yes) to suppress the display, or to N (for no) to display the item ERRCODE and the text of the error message. If the component does not set the item, the default is N.
PARAMS-OFFSET	Offset in item PARAMS-BUFFER for the parameters of the error or warning message. The value is -1 if PARAMS-COUNT equals 0.
PARAMS-COUNT	Number of parameters returned for the error or warning message.
ARRIVAL-SEQ	Sequence in which the error was set. The value is set to -1 when the SQLCA is initialized. For internal use only.
SQLCODEA	Redefinition of ERRORS-ALL.
SQLCODE	First error or warning code. SQLCODE redefines ERRCODE (1). Use this data item to test if errors or warnings occur. Use the SQLCA_DISPLAY2_, SQLCA_TOBUFFER2_, and SQLCAFSCODE procedures to obtain detailed information.

Table 9-1. SQLCA Structure Fields (page 3 of 3)

Field Name	Description
FILLER	Filler item.
PARAMS-BUFFER	Information about warnings and errors. PARAMS-BUFFER-LEN is the length in bytes. You can use the SQLCA_DISPLAY2_ procedure to read information returned to this buffer. Each parameter is stored as a string of ASCII printable characters terminated by a byte containing binary 0.
SRC-NAME-BUFFER	Name of the program source file. This buffer contains the name of a file specified with a SOURCE directive. When the SQL statement source code is in the input, this buffer is empty.

Using Constraints to Check for Errors

SQL/MP supports constraints to protect the integrity of base tables. A constraint is a condition that must be met before data is added to a row in the base table to which the condition applies. For example, a constraint might restrict employee numbers to the range between 0001 and 9999. If you try to enter a value outside this range, SQL/MP returns an error when it executes the UPDATE or INSERT statement.

You can create or drop constraints at any time. Creating or dropping a constraint, however, causes the system to invalidate all SQL program files that use the underlying table. Ensure that these files are explicitly SQL compiled to avoid automatic recompilation every time a program runs.

Note. When you add a constraint, SQL/MP checks all rows in the table. For large tables, the CREATE CONSTRAINT operation can cause performance problems.

Constraints are also a replacement for program code for all programs that refer to a table to which constraints apply. For example, constraints can be used to establish value ranges for columns, true or false conditions, and so forth.

Consider this example:

```
EXEC SQL CREATE CONSTRAINT MGRNUM_CONST ON DEPT
  CHECK MANAGER BETWEEN MIN_MANAGER AND MAX_MANAGER END-EXEC.
```

In this example, the constraint on the DEPT table restricts the value of the MANAGER column to the range shown. A program does not have to check the value for each insertion or update to this table. SQL/MP ensures the value is within the range. If the value is not within the range, SQL/MP returns an error message and aborts current TMF transactions. Constraints are a database protection mechanism. A program does not have to perform the checks to avoid corrupting the database.

This constraint in this example examines two columns within the same row and ensures that the employee termination date is equal to or greater than the date of hire:

```
EXEC SQL CREATE CONSTRAINT VALIDATE_CONST ON EMPLOYEE
  CHECK TERM_DATE >= HIRE_DATE END-EXEC.
```

Before coding constraint-checking logic into requester programs, consider that constraints can change over time and that such changes mean recoding the requesters with new checks to match the changed constraints. Also, consider that operators learn quickly what causes data-entry errors and can be trained during system testing to avoid such errors. Constraints are a database protection mechanism, not a substitute for operator training and proper system documentation.

Instead of allowing constraints to check for errors, you can code a Pathway requester to check entered data for compliance with the constraints. If certain data is prone to operator-entry error, a constraint detects the error only when the server attempts to update the database. A requester program can detect such operator-entry errors before sending the data to the server.

When the requester checks for errors rather than letting constraints do the checking, error message traffic decreases between the requester and server and between the server and disk process, and performance improves. Also, if there are many constraints on the same table, the server might find it difficult to determine which constraint caused which error, making it difficult to return a specific message to the terminal. It is easier for a requester to determine which entered value caused an error.

System designers must evaluate each application to determine whether or not to code constraint logic in programs to detect errors. When making this decision, consider these points:

- Requester checks that parallel constraints on entered data are very efficient. However, such checks require extensive coding and must be recoded to match changes to constraints.
- Constraints protect the database from operator error in applications and from update error in SQLCI. However:
    - Constraint checks are less efficient because they increase the message traffic.
    - When multiple constraint errors occur in a program, it is difficult to determine which entered value violated which constraint.
- Server checks on entered data are sometimes required (for example, if the result of the check affects subsequent processing).

Displaying and Storing Errors and Warnings

When you display SQL errors and warnings, you must consider where to display or store the error. For example, do you send the error message to a terminal, a file, or an SQL table? Usually, sending the error message to an SQL table is the preferred method.

Terminal

Using HOMETERM is not always advisable because more error messages could be generated than HOMETERM can handle. For example, an incorrect table name in a DEFINE could make the table unavailable to a Pathway system. This error would cause every server program to receive an SQL error each time it referred to that table.

Also, consider that other components of the system, such as TMF, might be using HOMETERM. If HOMETERM is a printer, printing might be delayed, or if HOMETERM is a terminal, messages can be lost when text exceeds the size of the screen buffer.

Another consideration is that access to HOMETERM might not be easy for an end user of the application. Suppose that the end user is notified that something failed, but the error text is sent to HOMETERM. The end user might find it difficult to determine the exact reason for the failure to correct it. If many errors are routed to HOMETERM, the database administrator might find it difficult to match the error to the user. Processing so many errors can also be time consuming.

Errors File

An option in SQLCA_DISPLAY2_ allows errors and warnings to be routed to a file. You might consider this alternative, but it has the disadvantage that you must write a program to extract the errors and warnings from the file.

You could write a separate process to monitor the file and print the messages as they arrive. The process could search the file for specific errors. If the prefix and suffix of the messages contain unique data, such as the terminal ID and a timestamp, the process could use this data to locate and report on specific errors.

SQL Errors Table

Instead of sending errors to a file, send the errors to a nonaudited SQL table. Using an SQL table makes it easy to retrieve the error information from the table with SQLCI. Users or database administrators can formulate queries to answer their questions and to monitor the systems. The report writer facilities of SQLCI help you produce complete reports.

You use a nonaudited table so you can keep a log of messages generated by backed out transactions. If you use an audited table, messages are lost when they are associated with transactions backed out by TMF. Another advantage is that inserts to a nonaudited table do not incur TMF overhead.

Your SQL error routine could include a call to SQLCA_TOBUFFER2_ to send the error messages to a storage buffer and a call to SQLCAFSCODE to obtain any file system errors associated with the SQL error. Then, you could code your program to insert a row into the table.

You can create the error table as a key-sequenced table with a timestamp as the primary key and indexes on other useful information. Although you could create a relative table that allows rows to be appended at the end, or an entry-sequenced table, a key-sequenced table with alternate keys offers the best searching capability.

Suppose that TABLEX and its index XTABLX are created:

```
CREATE TABLE TABLEX (
   ERRDATE     NUMERIC (6)       NO DEFAULT     ,
   ERRTIME     NUMERIC (6)       NO DEFAULT     ,
   TERMID      NUMERIC (6)       NO DEFAULT     ,
   SQLCODE     NUMERIC (4)       NO DEFAULT     ,
   FSCODE      NUMERIC (4)       SYSTEM DEFAULT ,
   TEXT1       CHARACTER (240)   SYSTEM DEFAULT ,
   TEXT2       CHARACTER (240)   SYSTEM DEFAULT ,
   TEXT3       CHARACTER (240)   SYSTEM DEFAULT ,
   TEXT4       CHARACTER (240)   SYSTEM DEFAULT ,
   PRIMARY KEY ( ERRDATE ASC,  ERRTIME ASC ) )
NO AUDIT
SECURE "NNNN"

CREATE INDEX XTABLX ON TABLEX (
   TERMID,
   SQLCODE,
   FSCODE )
```

The timestamp (ERRDATE, ERRTIME) is the primary key of the table. The terminal ID, the SQL code, and any file system code are the composite alternate index for the table. For each error, you write one row to the table. To collect the information for this table, the program can perform these steps:

1. Retrieve the timestamp, terminal ID, and SQLCODE for the first four columns.
2. Call SQLCAFSCODE to retrieve FSCODE (or zero) for the fifth column.
3. Call SQLCA_TOBUFFER2_ to retrieve the error message text for the last four columns.

If you send error information to an SQL table, you might consider setting up a help desk to provide end users with more information about errors returned by the application. (A help desk is a person or group of people responsible for helping end users with their problems.) When individual users require more explicit information about errors, the help desk staff can query the error table with SQLCI.

A query from TABLEX could specify the user terminal ID and the date and approximate time of the error (for example, between 2:00 p.m. and 3:00 p.m. June 8, 1996) as shown:

```
SELECT * FROM TABLEX
         WHERE ERRDATE = "960608"
           AND TERMID  = "012300"
           AND ERRTIME BETWEEN "140000" AND "150000"
```

Other queries could look for specific SQL errors (such as -8300) using the SQLCODE column, or specific file-system errors (such as 11) using the FSCODE column.

On a periodic basis, you can archive the SQL error table with the COPY or DUP utilities available through SQLCI, and you can clear the current table with the PURGEDATA utility. In this way, you can restrict the table to contain errors for a specific period of time, such as one week.

Each Monday, for example, you could use DUP to copy TABLEX to a TABLEX1, and then clear TABLEX before activating the Pathway system. The errors found this week and last week are available. Only a minimum amount of program code is needed to support this facility.

Selective Error Reporting

SQL/MP returns an error condition for many different errors. Some of these are true errors, others are not. For example, -8227 indicates the user is trying to insert a duplicate row (a mistake), and -8300 indicates a file-system error (a true error).

Both conditions cause the WHENEVER SQLERROR procedure to be executed. You could assume that the first condition should be reported only to the terminal as a mistake, whereas the second condition should be reported as an error to database administration.

Mistakes can also be considered as anticipated errors because the program can anticipate them. A record not found or an end-of-file condition, like the duplicate record condition, are errors that can be anticipated and reported to the terminal, not logged to an error table.

Here are some other examples of anticipated errors:

Error Number	Description
4028	Value for column x has an incompatible data type.
8230	Zero rows returned by subquery.
8233	Constraint number x violated (possibly a special situation).
8405	Decimal data encountered with some nonnumeric digits.

You could examine all the SQL error codes and decide which should be reported as true errors and which should be reported to a terminal as anticipated errors. This task is not recommended because the number of errors is large. Also, the application requirements might vary. For example, it might depend on your application requirements whether you consider timeout errors to be anticipated errors or true errors to be reported to a log file.

You can develop a common routine to test such conditions and to react accordingly. If you develop such a common routine, consider having two such routines: one to use for testing and one for production.

Many errors are programming errors, such as 8225 and 8226, which are cursor declaration errors. Other errors are run-time errors. To facilitate testing, reply to the terminal when anticipated errors are found during development and also log them as errors. Use this strategy to determine which SQL statements failed and why.

When you go to production, you no longer want to log the anticipated errors. As a result, the number of errors logged during production is less than the total number logged during development. The production routine should check for anticipated errors and reply to the terminal only.

Summary of Error Processing Recommendations

This list summarizes the recommendations described in this subsection for processing errors.

- Use WHENEVER directives to test for SQL errors, warnings, and other conditions. Each WHENEVER directive should refer to a common error handling routine.
- Develop common error handling routines that can be saved in a source library and copied with a COBOL COPY command or an COBOL SOURCE directive.
    - Include a call to the SQLCA_TOBUFFER2_ or to SQLCA_DISPLAY2_ procedure in any error routine that logs or displays the errors.
    - Consider using two sets of common error handling routines: one for development and one for production. In the development routine, specify a breakpoint or a call to the Inspect program.
- Establish meaningful messages to be returned to the operator, such as:
    - A not-found condition returns Requested Data Does Not Exist.
    - A duplicate condition returns Data Cannot Be Inserted - Already Exists.
    - A file-system error returns File-System Error *error* occurred. Use SQLCAFSCODE to determine the file-system error number.
    - An SQL error returns SQL Error *error* occurred.
- Create an SQL table to save the error messages. Use SQLCA_TOBUFFER2_ to return the error code, message text, program line number, and other information you save in the error table:
    - Put a timestamp, terminal ID, SQL error, and file-system error in the first columns of the table row to be used as search criteria for later retrieval.
    - Write the error information to the last columns of the table row.
    - Specify parameters in the calls to SQLCA_TOBUFFER2_ to reduce the number of error lines generated for each error by suppressing statistics and the internal error location and by setting the prefix to a single blank character.
    - Specify parameters in the calls to SQLCA_TOBUFFER2_ to generate error lines of 80 characters each to match the line length that SQLCI displays.
    - Define the table columns that are to receive the error information as multiples of 80, but not greater than 240, characters. This size prevents wraparound because SQLCI displays large columns as 80-character lines and prevents truncation because SQLCI displays a maximum of 255 characters per column.
    - In summary, retrieve the error information in 80-byte error lines and define two, three, or four text columns of 240 bytes each in the SQL error table.
    - Use SQLCI to query and display the error table.

- If you route errors directly to HOMETERM by using the SQLCA_DISPLAY2_ routine, save the errors in an SQL error table also. This table can be used as an easily retrievable record of errors.
- Archive the error table. Using the SQL utilities, make a copy of the table and purge the current data. Retain the copy for future reference. The error table might also be useful for reporting errors to your service provider.
- Distinguish between true errors, such as file-system errors, and anticipated errors, such as an attempt to insert a duplicate row.
  - Save all errors, including anticipated errors, in the development routines.
  - Save only selected (true) errors in the production routines.
- Consider establishing a help desk to assist end users in resolving errors. A help desk staff can query the error table to determine which error is associated with which user. In some situations, an end user might query the table directly if the user has access to SQLCI and the data is understandable.
- A database administrator might also want to query the table for other errors such as file-system errors (SQL error 8300).
- A database administrator might generate reports to answer these questions:
  - How many errors occurred today?
  - How often does a user or group of users violate constraints?
  - How often does a user or group of users attempt to insert duplicate rows?

Returning Performance and Statistics Information

SQL/MP returns performance and statistics information to the SQL statistics area (SQLSA) after the execution of these DML statements:

- An INSERT, UPDATE, and DELETE statement
- A SELECT statement with the INTO clause for a host variable
- An OPEN, CLOSE, or FETCH statement for a cursor operation that has a SELECT statement specified in the DECLARE CURSOR statement

For dynamic SQL operations, SQL/MP returns information in the SQLSA structure for these statements:

- Each PREPARE statement, which includes information about input parameters, output columns, and the length of the input and output names buffer
- Each DESCRIBE statement, including information about input parameters, output columns, the names buffer, and the collation buffer
- Each DESCRIBE INPUT statement, including information about input parameters, output columns, and the names buffer

The SQLSA structure is undefined after the execution of a DSL, DDL, DCL, or transaction control statement.

Use this syntax for the INCLUDE SQLSA directive to declare the SQLSA in the Data Division of your program (but not in a Declare Section):

```
EXEC SQL INCLUDE SQLSA END-EXEC.
```

Follow these guidelines when you declare and use an SQLSA structure:

- You might need to specify an INCLUDE STRUCTURES directive with the version of the SQLSA structure you require. If you do not first specify this directive, SQL/MP generates version 2 SQL structures by default.
- Use the SQLSADISPLAY system procedure to write information from the SQLSA structure to a file or terminal. For information about SQL system procedures, see Section 5, SQL/MP System Procedures.
- A new statement resets the SQLSA structure fields. If you are using a value elsewhere in your program, you might need to save the value immediately after the statement executes (or declare more than one SQLSA structure).
- Each FETCH statement resets the SQLSA structure. To calculate statistics for a cursor, declare accumulator variables for the required statistics. Then add values from the SQLSA fields to the accumulator variables after each FETCH operation.

Example 9-3 on page 9-23 shows an SQLSA structure generated in a COBOL program by the INCLUDE SQLSA directive. (For the version 1 and version 2 SQLSA structures, see Appendix D, Converting COBOL Programs.)

Example 9-3. SQLSA Structure

```
01 SQLSA.
  02 EYE-CATCHER          PIC X(2)   VALUE "SA".
  02 VERSION              PIC S9(4) COMP VALUE 0.
  02 DML.
    03 NUM-TABLES           PIC 9(4) COMP VALUE 0.
    03 STATS OCCURS 16 TIMES.
      04 TABLE-NAME           PIC X(24) VALUE SPACES.
      04 RECORDS-ACCESSED     PIC S9(9) COMP VALUE 0.
      04 RECORDS-USED         PIC S9(9) COMP VALUE 0.
       04 DISC-READS           PIC S9(9) COMP VALUE 0.
       04 MESSAGES             PIC S9(9) COMP VALUE 0.
       04 MESSAGE-BYTES        PIC S9(9) COMP VALUE 0.
       04 WAITS                PIC S9(4) COMP VALUE 0.
       04 ESCALATIONS          PIC S9(4) COMP VALUE 0.
       04 SQLSA-RESERVED       PIC X(4)  VALUE SPACES.
   02 PREPARE REDEFINES DML.
     03 INPUT-NUM            PIC 9(4) COMP.
     03 INPUT-NAMES-LEN      PIC 9(4) COMP.
     03 OUTPUT-NUM           PIC 9(4) COMP.
     03 OUTPUT-NAMES-LEN     PIC 9(4) COMP.
     03 NAME-MAP-LEN         PIC 9(4) COMP.
     03 SQL-STATEMENT-TYPE   PIC 9(4) COMP.
       88 SQL-STATEMENT-SELECT  VALUE 1.
       88 SQL-STATEMENT-INSERT  VALUE 2.
       88 SQL-STATEMENT-UPDATE  VALUE 3.
       88 SQL-STATEMENT-DELETE  VALUE 4.
       88 SQL-STATEMENT-DDL     VALUE 5.
       88 SQL-STATEMENT-CONTROL VALUE 6.
       88 SQL-STATEMENT-DCL     VALUE 7.
       88 SQL-STATEMENT-GET     VALUE 8.
     03 OUTPUT-COLLATIONS-LEN  PIC 9(4) COMP.
```

Table 9-2 describes each SQLSA structure field.

Table 9-2. SQLSA Structure Fields (page 1 of 2)

Field Name	Description
EYE-CATCHER	Identification field. The compiler sets EYE-CATCHER to SA.
VERSION	Current version of the SQLSA.
DML	Group item where statistics for DML statement execution are returned.
NUM-TABLES	Number of tables accessed by a DML statement. The maximum number is 16.
STATS	Array containing NUM-TABLES valid entries, one for each table accessed.
TABLE-NAME	Guardian internal file name of the table accessed.
RECORDS-ACCESSED	Number of records accessed in the corresponding table.

Table 9-2. SQLSA Structure Fields (page 2 of 2)

Field Name	Description
RECORDS-USED	Number of records altered or returned.
DISC-READS	Number of disk reads and writes.
MESSAGES	Number of messages sent to the disk process.
MESSAGE-BYTES	Number of bytes sent in all the messages sent to the disk process.
WAITS	Number of lock waits or timeouts.
ESCALATIONS	Number of times record locks are escalated to file locks.
SQLSA-RESERVED	Reserved.
PREPARE	Group item where statistics for a PREPARE statement are returned.
INPUT-NUM	Number of input parameters in the prepared statement.
INPUT-NAMES-LEN	Length of the buffer required to contain names of the input parameters.
OUTPUT-NUM	Number of output variables (host variables or SELECT columns) in the prepared statement.
OUTPUT-NAMES-LEN	Length of the buffer required to contain names of the output variables.
NAME-MAP-LEN	Length of the buffer for name maps. A value is returned in this item only if the name map will be used or saved for later use. The system stores in name maps the context of DEFINEs, default subvolume, and so forth.
SQL-STATEMENT-TYPE	Type of statement being prepared. The values can be:

Level 88 Item Name	Value	SQL Statement or Directive
SQL-STATEMENT-SELECT	1	Cursor SELECT
SQL-STATEMENT-INSERT	2	INSERT
SQL-STATEMENT-UPDATE	3	UPDATE
SQL-STATEMENT-DELETE	4	DELETE
SQL-STATEMENT-DDL	5	DDL statement
SQL-STATEMENT-CONTROL	6	Run-time CONTROL TABLE
SQL-STATEMENT-DCL	7	LOCK, UNLOCK, FREE RESOURCES
SQL-STATEMENT-GET	8	GET VERSION...

Field Name	Description
OUTPUT-COLLATIONS-LEN	Length of the output collation buffer.

10 Dynamic SQL Operations

Dynamic SQL is useful if you do not know all or part of an SQL statement before run time. In this case, you cannot program the statement into your application. The program must be constructed at run time. For example, you might want to process SQL statements from a user or accept a statement generated by an application on a personal computer.

When you use dynamic SQL, you can construct or obtain SQL statements at run time and then compile them and execute them. A set of SQL statements known as dynamic SQL statements support this capability. When you use these statements, you can submit other SQL statements (such as UPDATE or DELETE requests) dynamically at run time.

Table 10-1 summarizes dynamic SQL statements. These statements are described, with examples, in the remainder of this section. Your use of a specific statement depends on your application requirements.

Table 10-1. SQL Statements Used for Dynamic SQL Operations

Statement	Description
DESCRIBE INPUT	Obtains information about input parameters.
DESCRIBE	Obtains information about output parameters (SELECT columns).
PREPARE	Dynamically compiles a statement—such as one requested by a user or constructed by the program—that was not known until run time.
EXECUTE	Executes a prepared statement.
EXECUTE IMMEDIATE	Executes a statement without preparing it first.
DECLARE CURSOR	Defines a cursor and associates the cursor with a statement name or host variable name. (Used only with SELECT statements.)
OPEN	Opens a cursor. The dynamic form includes a USING clause that allows you to provide values for dynamic parameters. (Used only with SELECT statements.)
FETCH	Fetches data through the cursor. (Used only with SELECT statements.)
RELEASE	Deallocates space for a dynamic SQL statement prepared from a host variable. (Used only with SELECT statements.)
CLOSE	Closes the cursor. (Used only with SELECT statements.)

Topics include:

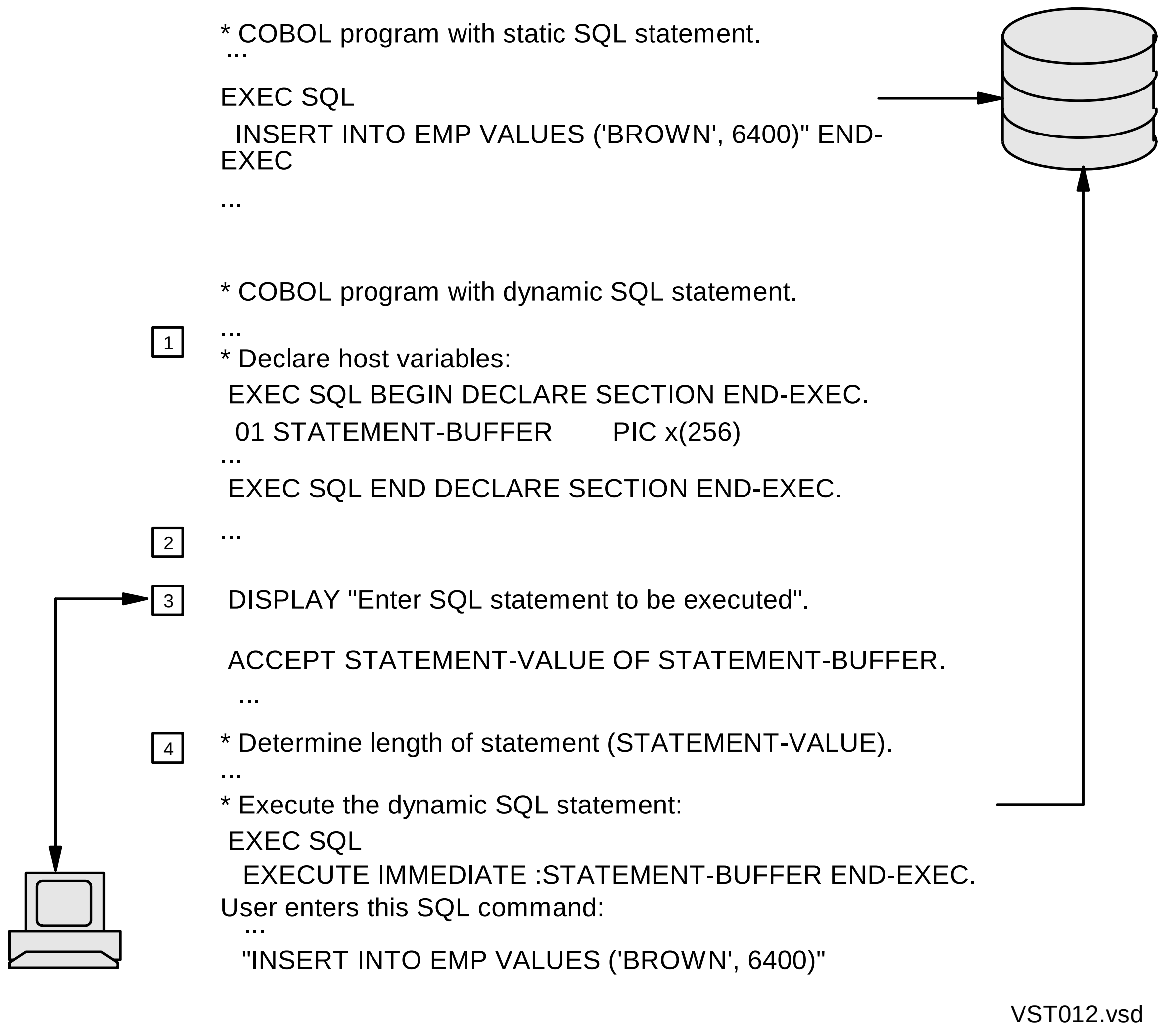
- Developing a Dynamic SQL Application on page 10-9
- Constructing a Server that Interfaces With Pathway on page 10-35
- Sample Dynamic SQL Program on page 10-37

Using Dynamic SQL

Figure 10-1 shows part of a COBOL program (top) that contains an embedded static SQL statement and a part of another COBOL program (bottom) that accepts a request from a user to execute a dynamic SQL statement. The steps used for the dynamic SQL statements are:

1. Define a character string host variable to hold the dynamic SQL statement.
2. Display a prompt to the user, requesting the SQL statement.
3. Accept the SQL statement into the host variable and determine its length.
4. Execute the SQL statement using an EXECUTE IMMEDIATE statement.

Figure 10-1. Static and Dynamic SQL Programs



```
* COBOL program with static SQL statement.
 ...
EXEC SQL
  INSERT INTO EMP VALUES ('BROWN', 6400)" END-
EXEC
...

* COBOL program with dynamic SQL statement.
...
* Declare host variables:
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 STATEMENT-BUFFER        PIC x(256)
...
 EXEC SQL END DECLARE SECTION END-EXEC.
...
 DISPLAY "Enter SQL statement to be executed".

 ACCEPT STATEMENT-VALUE OF STATEMENT-BUFFER.
   ...
* Determine length of statement (STATEMENT-VALUE).
...
* Execute the dynamic SQL statement:
 EXEC SQL
   EXECUTE IMMEDIATE :STATEMENT-BUFFER END-EXEC.
User enters this SQL command:
   ...
   "INSERT INTO EMP VALUES ('BROWN', 6400)"
```

Dynamic SQL can be less efficient than static SQL because more work is deferred until run time. This table compares stages of processing for static and dynamic SQL:

SQL Operation	**Static SQL Operation**	**Dynamic SQL Operation**
Parse the SQL statement	At compile time (or as invoked by COBOL)	Run time
Validate the statement	At compile time (or as invoked by COBOL)	Run time
Optimize the statement	SQLCOMP	Run time

SQL Operation	Static SQL Operation	Dynamic SQL Operation
Generate an application plan	SQLCOMP	Run time
Store the application plan	SQLCOMP	Not applicable
Execute the statement	Run time	Run time

Uses for Dynamic SQL

Dynamic SQL can be very useful if an application requires:

- Flexibility to construct SQL statements at run time (for example, an interactive interface similar to SQLCI but designed for an inexperienced user).
- Flexibility to defer an association with a database until run time (for example, an application that switches between several copies of identical databases). For this application, use a dynamic SQL program with run-time TACL DEFINEs.
- Restriction of access to data in a table. For example, the program might code an UPDATE statement for certain columns in a table but allow the user to enter any selection criteria (WHERE clause) at run time.
- Client-server support with deferral of definition of SQL statements until run time. For example, the user of an application on a personal computer wants to manipulate data in a database on a host system. Such an application cannot use SQLCI. The user formulates an SQL statement on the personal computer, and the application sends the statement to a server process on the host system over Multilan or another communications protocol.

If you plan to execute an SQL statement only once, execute the statement dynamically and save any memory that would have stored the execution plan.

A dynamic SQL application can accept statements directly from the user or through a screen interface like Pathway, or the program can build the statements with little or no user input. The application might process an entire SQL statement, or it might process only part of a statement (such as the WHERE clause) and explicitly code the remainder of the statement in the program. For more information about using dynamic SQL within a Pathway server, see Constructing a Server that Interfaces With Pathway on page 10-35.

A program that uses dynamic SQL to process input directly from a user can be similar to SQLCI, requiring the user to understand SQL syntax to formulate a complete SQL statement. The statement can contain input parameters. If it does, the program can prompt the user for the parameter values.

You can also write a program for direct user input so that the user does not have to understand SQL syntax. In this case, the program prompts the user for the necessary values (or displays a screen on which the user enters the values) and then constructs the SQL statement by concatenating these values to known syntax elements.

For example, a program can handle any CREATE TABLE statement by concatenating the string “CREATE TABLE” and punctuation (for example commas and colons) to the table name, column names, data types, and options entered by a user and stored in local variables. The program user sees only a series of prompts, such as ENTER THE TABLE NAME, ENTER THE FIRST COLUMN NAME, and so forth.

Determining When to Use Dynamic SQL

If you do not know the whole text of an SQL statement at development time but there are only a few alternatives, you might want to program the alternatives into your application. Otherwise, for applications that require greater flexibility, use dynamic SQL statements.

Features of Dynamic SQL

When you write a program that uses dynamic SQL, you use many of the same SQL statements as you would in static SQL. You can perform most of the same operations using dynamic SQL statements that you perform with static SQL statements. You can use DDL, DML, and DCL statements in both modes.

The difference between the two modes is that all or part of a dynamic SQL statement is obtained from the user or generated by your program, stored in a character host variable, compiled, and executed at run time. With dynamic SQL statements you must perform some additional operations, such as building descriptors for host variables, that the HP COBOL compiler performs for you when you use static SQL statements.

After compilation, NonStop SQL/MP executes statements in the same way whether they are dynamic SQL statements or static SQL statements. SQL/MP places results of dynamic SELECT statements into output parameters. You can use the DESCRIBE statement to obtain information about the output parameters (also called select columns).

Processing Database Requests

Use these SQL statements to execute an SQL database request dynamically:

- Use the EXECUTE IMMEDIATE statement to compile and execute a statement whose text is contained in a host variable. Execution is performed in one step, which is very useful if you plan to execute an SQL statement only once. SQL/MP does not store any information about the statement.
- Use the PREPARE and EXECUTE statements to prepare a statement for execution, save the information about the statement, and then execute the statement as often as needed with these considerations:
    - The PREPARE statement associates a statement name with the SQL statement specified by an SQL identifier and dynamically compiles an SQL/MP statement. After a statement is prepared, the program can execute the statement with the EXECUTE statement or (for SELECT statements) with a cursor.

When you prepare a statement, the database environment generates an access plan and a description of the result set. Preparation is useful if you want to execute a statement multiple times with the previously generated access plan to minimize processing overhead.

- The EXECUTE statement executes a previously prepared dynamic SQL statement. You can use EXECUTE for any DDL, DML, or DCL statement except SELECT. (Use a cursor to process a SELECT statement.)

Applications must process SELECT and nonSELECT statements differently. You do not necessarily know what type of statement your program is processing. The statement could, for example, be a SELECT, UPDATE, or DELETE statement. To determine whether a statement is a SELECT statement, check the SQLSA. SELECT and cursor statements always have at least one returned value.

This example functions the same as the example in Figure 10-1 on page 10-3, except that it uses the PREPARE and EXECUTE statements instead of EXECUTE IMMEDIATE:

```
(define the host variable)
 DISPLAY "Enter statement to be executed:".
*  User enters INSERT statement here
 ACCEPT VAL of STATEMENT-BUFFER
(determine length of VAL)

 EXEC SQL PREPARE S1 FROM :STATEMENT-BUFFER  END-EXEC.
...

 EXEC SQL EXECUTE S1  END-EXEC.

  (Insertion performed)
```

Using Parameters

A parameter is an SQL identifier that serves as a place holder in a dynamic SQL statement for a value substituted when the statement executes. (You can also use a parameter with SQLCI.) You use parameter markers, with names preceded by question marks (?), in place of each parameter. For example:

```
SELECT * FROM MONTHLY_PAYROLL WHERE NAME = ?NM.
```

The SQL statement is compiled without the actual input values, which are substituted for the parameter when the SQL statement executes. For the syntax for a parameter, see the *SQL/MP Reference Manual.*

Input Parameters

A dynamic SQL statement can contain input parameters, which allow your program to construct SQL statements at run time. Input parameters convey data from the host program to the SQL statement. They might denote criteria to be used in a WHERE clause, values to be inserted into the database, or values used to update or delete database records.

Input parameters function in much the same way as host variables in embedded SQL. An input parameter can appear in an SQL expression wherever a constant can appear. You specify input parameters in a statement as either a question mark (?) or a question mark plus a name (?VAL). Before execution, you assign values to the place held by each parameter. Unlike static SQL, dynamic parameters do not require length or data type definition before program compilation.

When you submit an SQL statement dynamically, you do not necessarily know the number or types of parameters. To obtain information about input parameters and obtain pointers to the names of the input values, use the DESCRIBE INPUT statement. You can obtain the number of input values from the SQLSA.

Output Parameters

SQL/MP returns data to your program through output parameters, which are user-defined areas in the program. Output parameters can be host variables or individual data buffers to which the program’s output SQLDA structure points. Output parameters usually contain columns returned from a SELECT operation.

When you submit an SQL statement dynamically, you do not always know the number or types of parameters. To obtain information about the output parameters, use the DESCRIBE statement with an output SQLDA structure. You can obtain the number of output values from the SQLSA.

Using a Parameter List

To ensure a one-to-one correspondence between a parameter list and the host variables you use to supply values for the parameters, use unnamed parameters. If duplicate parameter names appear in a statement, the names require a value for only the first occurrence, and the duplicate occurrences receive the same value.

For example, suppose that this UPDATE statement is stored in the host variable UPDATE-STATEMENT:

```
UPDATE ATABLE SET COL1 = ?A, COL2= ?A, COL3 = ?B
```

A PREPARE statement prepares the statement in the host variable named UPDATE-STATEMENT:

```
EXEC SQL
PREPARE EXECUTE-STATEMENT FROM :UPDATE-STATEMENT
END-EXEC.
```

To supply values for the UPDATE statement at run time, the program uses the two host variables HOST-VAR1 and HOST-VAR2:

```
EXEC SQL
  EXECUTE EXECUTE-STATEMENT USING :HOST-VAR1, :HOST-VAR2
END-EXEC.
```

The value in HOST-VAR1 is used for both instances of parameter ?A. The value in HOST-VAR2 is used for parameter ?B. For three host variables, SQL/MP uses the value in the first host variable for both occurrences of parameter ?A. The value in the second host variable is used for parameter ?B, but the value in HOST-VAR3 is unused.

For example, in this statement, SQL/MP uses the value in HOST-VAR1 for both occurrences of parameter ?A and the value in HOST-VAR2 for parameter ?B. The value in HOST-VAR3 is ignored.

```
EXEC SQL
  EXECUTE EXECUTE-STATEMENT USING USING :HOST-VAR1,
                                        :HOST-VAR2,
                                        :HOST-VAR3
END-EXEC.
```

△ **Caution.** If you use the same parameter name more than once in a statement, SQL/MP gives each duplicate occurrence of the parameter the same data type, length, and other attributes as the first occurrence. Therefore, data can be lost in some cases.

For example, during the execution of an INSERT statement, a parameter gets the same data type and attributes as the column into which the parameter's value is first inserted. If the parameter value is truncated to fit into the column, the values of any duplicate occurrences of the parameter are also truncated, even if a column is large enough to hold the complete value.

Using Parameters in a Loop

Parameters are often used when a dynamic SQL statement is executed repeatedly with different input values. In these examples, a dynamic SQL statement uses a parameter. Because the user of this program can enter any SQL statement, the program does not have information about the statement during compilation. The DEFINE **=**PARTS represents the PARTS table.

1. A user enters this SQL statement from a terminal:
2. UPDATE =PARTS SET PRICE = ?P
3. The program copies the statement into the host variable INTEXT.
4. The program uses the PREPARE and DESCRIBE INPUT statements to put a description of the parameter in the SQLDA structure IN-SQLDA and to put the name of the parameter in NAMESBUF, the input names buffer**.** The prepared statement is named S1.

   ```
   EXEC SQL PREPARE S1 FROM :INTEXT  END-EXEC.
   EXEC SQL
     DESCRIBE INPUT S1 INTO :IN-SQLDA NAMES INTO :NAMESBUF
   END-EXEC.
   ```

5. The program enters a loop and prompts the user to supply values for successive execution of the statement:

   ```
   * BEGINNING OF LOOP

   * PROMPT THE USER FOR A VALUE USING THE
   * PARAMETER NAME FROM THE NAMES BUFFER.
   ...
   ```

```
* STORE THE VALUE IN AN INPUT BUFFER
* POINTED TO BY IN-SQLDA.

* EXECUTE THE STATEMENT USING EACH SUCCESSIVE VALUE.

 EXEC SQL EXECUTE S1 USING DESCRIPTOR :IN-SQLDA END-EXEC.

* END OF LOOP
```

Using Indicator Parameters

A program uses an indicator parameter to indicate that a null value was entered for a parameter. The indicator parameter follows the parameter in the SQL statement as shown in the next example:

```
INSERT INTO =employee VALUES (1000, ?p INDICATOR ?i );
```

If a user enters a null value for ?P, the program should set ?I to a value less than zero. If a user enters a nonnull value for ?P, the program should set ?I to 0. Both ?P and ?I are in the names buffer, so the program can prompt the user for a null value.

Developing a Dynamic SQL Application

The simplest type of dynamic SQL program does not have any input parameters or output parameters. This type of program processes statements such as CREATE TABLE or DROP INDEX, which do not require input parameters.

These features add complexity to dynamic SQL programs:

- Support for input or output parameters. For example, a SELECT statement can add complexity to a dynamic SQL program. The program cannot determine the number of SELECT columns and input parameters until run time.
- Support for null values and indicator parameters and variables.

This subsection includes information about how to support parameters and null values in dynamic programs. The topics might not all apply to your application.

In general, the steps in a dynamic SQL application are:

1. Declare a host variable for the SQL statement to be submitted.
2. Declare the SQLCA and SQLSA data structures.
3. If you plan to support input or output parameters in the SQL statement, perform these steps:
    a. Declare the SQLDA, names buffer, if desired, and collation buffer, if desired, to describe the parameters.
    b. Define buffers for parameter values of different data types.

4. If you plan to execute the statement more than once, or if your statement includes input or output parameters, prepare the SQL statement to compile the statement dynamically and assign it a statement name.
5. Determine whether there are parameters in the SQL statement by examining the SQLSA. Then use the DESCRIBE INPUT and DESCRIBE statements as needed.
6. If there are parameters in the statement, move their descriptions into an SQLDA, set up the SQLDA structure to point to the storage for variables referenced by the query, and initialize appropriate field values.
7. If there are input parameters in the statement, prompt the user for input. You can use the names buffer to prompt the user. Depending on your situation, you might also want to handle null values on input.
8. Using the input SQLDA (if there were parameters), perform one of these database requests:
    - Execute the statement (for a statement that is not a SELECT statement). If you prepared the statement, use an EXECUTE statement. Otherwise, use an EXECUTE IMMEDIATE statement.
    - Process the SELECT statement by issuing a cursor FETCH statement.
9. Display the output. If necessary, handle null results in the output.

The next subsections discuss these steps. Examples illustrate a dynamic SQL program that handles any statement and allocates memory at run time. The examples use hard-coded cursor names and statement names (such as C1 and S1) to store the cursor name and statement name. When a program uses host variables, the program can dynamically compile multiple statements and make all the statements available for execution simultaneously.

These examples show several methods of using dynamic SQL statements but are not intended to represent either the most efficient or the only method to develop a particular application.

Declaring a Host Variable

In a Declare Section in the Data Division, declare a host variable to serve as the buffer or “container” for the SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
  01 STATEMENT-BUFFER  PIC X(256) VALUE SPACES.
EXEC SQL END DECLARE SECTION  END-EXEC.
```

Declaring the SQLCA and SQLSA Data Structures

In the Data Division, declare the SQLCA and SQLSA data structures using INCLUDE directives:

```
EXEC SQL INCLUDE SQLCA  END-EXEC.
EXEC SQL INCLUDE SQLSA  END-EXEC.
```

The SQLSA stores information about the statement. (In contrast, the SQLDA stores information about parameters.)

Defining Storage for Input and Output Parameters

This subsection describes how to allocate storage for parameters. The discussion starts with a description of the SQLDA structure and associated buffers.

SQLDA Structure, Names Buffer, and Collation Buffer

SQL/MP uses the SQL descriptor area (SQLDA) to return information about input parameters and output parameters in dynamic SQL statements. The SQLDA also stores pointers to these optional data buffers:

- The names buffer, which stores the names of input parameters or lists the names of selected columns.
- The collation buffer, which receives copies of any collations used by columns in the query.

You can use the SQLDA data structure in:

- A DESCRIBE INPUT statement to return information about input parameters.
- A DESCRIBE statement to return information about output columns or copies of any collations used by the columns
- The USING DESCRIPTOR clause of a FETCH statement to retrieve rows from an SQL table.
- The USING DESCRIPTOR clause of an EXECUTE statement to execute a dynamic SQL statement.

The sizes of the SQLDA and names buffer depend on the number of input parameters or output parameters referenced in dynamic SQL statements. Your program can have more than one SQLDA. In some cases, you can reuse the same structure for different SQL statements.

If there are input parameters in the dynamic SQL statement, you must have an input SQLDA to describe them. Similarly, if your program handles dynamic SELECT operations, declare an SQLDA to describe the output results (SELECT columns).

If your application prompts the user for parameter values or displays column names for output to the user, declare one or more names buffers.

SQLDA Contents

Table 10-2 describes the information stored in the SQLDA structure. Note that the SQLDA structure contains an SQLVAR record for each input parameter or output variable.

Table 10-2. SQLDA Structure Fields (page 1 of 2)

Field Name	Description	
EYE-CATCHER	An identifying field that a program must initialize. SQL/MP does not return a value to EYE-CATCHER.	
NUM-ENTRIES	The number of input or output parameters the SQLDA structure can accommodate.	
SQLVAR	Group item that describes input parameters or database columns. The DESCRIBE INPUT and DESCRIBE statements return one SQLVAR entry for each input parameter or each output variable.	
DATA-TYPE	The data type of the parameter. For the values used for each data type, see Table 10-3 on page 10-13.	
DATA-LEN	The DATA-LEN value depends on the data type:	
	Fixed-length character	The number of bytes in the string.
	Variable-length character	The maximum number of bytes in the string.
	Decimal numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item.
	Binary numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item (2, 4, or 8).
	Date-time or INTERVAL	Bits 0:7 contain one of these codes for the range of the field. Bits 8:15 contain the storage size of the item.

1	Year to Year	11	Year to Minute	20	Day to Minute
2	Month to Month	12	Year to Second	21	Day to Second
3	Day to Day	13	Year to Fraction	22	Day to Fraction
4	Hour to Hour	14	Month to Day	23	Hour to Minute
5	Minute to Minute	15	Month to Hour	24	Hour to Second
6	Second to Second	16	Month to Minute	25	Hour to Fraction
7	Fraction to Fraction	17	Month to Second	26	Minute to Second
8	Year to Month	18	Month to Fraction	27	Minute to Fraction
9	Year to Day	19	Day to Hour	28	Second to Fraction
10	Year to Hour				

Field Name	Description	
PRECISION	The PRECISION value depends on the data type:	
	Binary numeric	The numeric precision.
	Date-time or INTERVAL	Bits 0:7 contain the leading field precision. Bits 8:15 contain the fraction precision. If the FRACTION field is not included, bits 8:15 are 0.
	Character and VARCHAR	The character set ID (0 = UNKNOWN):

1	KANJI	101	ISO88591	104	ISO88594	107	ISO88597
12	KSC5601	102	ISO88592	105	ISO88595	108	ISO88598
		103	ISO88593	106	ISO88596	109	ISO88599

Table 10-2. SQLDA Structure Fields (page 2 of 2)

Field Name	Description
NULL-INFO	For input parameters, NULL-INFO contains a negative integer if the parameter permits null values. For output parameters, NULL-INFO contains a negative integer if the parameter can return a null value.
VAR-PTR	The extended address of the actual data (value of input parameter or database column). SQL/MP does not return VAL-PTR. Your program must initialize it to point to the input and output data buffer.
IND-PTR	The address of a flag that indicates whether a parameter or column is actually null. For input parameters, your program initializes IND-PTR to -1 if the user entered a null value. For output columns, SQL/MP initializes the location referenced by IND-PTR to -1 if the column value was null. If you program does not need to process null values, initialize IND-PTR to an invalid address.
CPRL-PTR	For input columns, CPRL-PTR is not set. For output columns, CPRL-PTR contains the address of the collation used by the column, if a collation was used. If a collation was not used for the output column, CPRL-PTR contains a negative integer.

This example shows a version 315 SQLDA structure. For a description of version 1 and version 2 SQLDA structures, see Appendix D, Converting COBOL Programs.

```
01 sqlda-name.
   05 EYE-CATCHER      PIC X(2) VALUE "D1".
   05 NUM-ENTRIES      PIC S9(4) COMP VALUE sqlvar-count.
   05 SQLVAR           OCCURS sqlvar-count TIMES.
      10 DATA-TYPE     PIC S9(4) COMP.
      10 DATA-LEN      NATIVE-2.
      10 PRECISION     PIC S9(4)  COMP.
      10 NULL-INFO     PIC S9(4)  COMP.
      10 VAR-PTR       PIC S9(9)  COMP VALUE -999999.
      10 IND-PTR       PIC S9(8)  COMP VALUE -999999.
      10 CPRL-PTR      PIC S9(9)  COMP VALUE -999999.
      10 RESERVED      PIC S9(9) COMP VALUE -1.
01 names-buffer        PIC X( names-buffer-length ).
01 collations-buffer   PIC X( collation-buffer-length ).
```

Table 10-3 lists the SQLDA DATA-TYPE values for each specific data type.

Table 10-3. SQLDA DATA-TYPE Values (page 1 of 3)

Value	Type of Data
Fixed-Length Character Data Types (0 – 63)	
0	Fixed-length single-byte character
1	Fixed-length single-byte character, upshifted
2	Fixed-length double-byte character

Table 10-3. SQLDA DATA-TYPE Values (page 2 of 3)

Value	Type of Data
VARCHAR Data Types (64 – 127)	
64	Variable-length single-byte character
65	Variable-length single-byte character, upshifted
66	Variable-length double-byte character
Numeric Data Types (128 – 191)	
130	16-bit signed (signed SMALLINT)
131	16-bit unsigned (unsigned SMALLINT)
132	32-bit signed (signed INT)
133	32-bit unsigned (unsigned INT)
134	64-bit signed (signed LARGEINT)
140	32-bit FLOAT (REAL)
141	64-bit FLOAT (DOUBLE PRECISION)
150	Unsigned DECIMAL
151	DECIMAL, leading sign separate (not SQL type)
152	DECIMAL, leading sign embedded
153	DECIMAL, trailing sign separate (not SQL type)
154	DECIMAL, trailing sign embedded (not SQL type)
Date-Time and INTERVAL Data Types (192 – 212)	
192	General Date-Time (DATETIME)
195	Year to Year (INTERVAL)
196	Month to Month (INTERVAL)
197	Year to Month (INTERVAL)
198	Day to Day (INTERVAL)
199	Hour to Hour (INTERVAL)
200	Day to Hour (INTERVAL)
201	Minute to Minute (INTERVAL)
202	Hour to Minute (INTERVAL)
203	Day to Minute (INTERVAL)
204	Second to Second (INTERVAL)
205	Minute to Second (INTERVAL)
206	Hour to Second (INTERVAL)
207	Day to Second (INTERVAL)
208	Fraction to Fraction (INTERVAL)
209	Second to Fraction (INTERVAL)

Table 10-3. SQLDA DATA-TYPE Values (page 3 of 3)

Value	Type of Data
210	Minute to Fraction (INTERVAL)
211	Hour to Fraction (INTERVAL)
212	Day to Fraction (INTERVAL)

Declaring the SQLDA Structure, Names Buffer, and Collation Buffer

To declare an SQLDA structure and associated buffers, use the INCLUDE SQLDA directive in the Data Division of your program (but not in a Declare Section). The syntax is:

```
INCLUDE SQLDA ( sqlda-name  [, sqlvar-count ]

            [, names-buffer, max-name-length                 ]

            [, release-option                                ]

            [, CPRULES collation-buffer, max-collation-size ]
)
```

sqlda-name

is the name of the SQLDA structure. *sqlda-name* must follow the conventions for COBOL names.

sqlvar-count

is either the maximum number of parameters (excluding indicator parameters) for which you expect to receive input values or, for an output SQLDA, the maximum number of output parameters. The default is 1. The HP COBOL compiler generates a separate SQLVAR structure within the SQLDA for each parameter.

names-buffer

is the COBOL record name of the names buffer. The INCLUDE SQLDA directive generates a template. For the names buffer, you must declare your own template.

max-name-length

is the maximum number of bytes you expect in a parameter name to be returned in a DESCRIBE or DESCRIBE INPUT statement. If you expect indicator parameters, double the value of *max-name-length*.

release-option

specifies the version of the SQLDA structure generated by the HP COBOL compiler. RELEASE1 specifies SQL/MP version 1, and RELEASE2 specifies SQL/MP version 2.

Note. Although the HP COBOL compiler supports the RELEASE1 and RELEASE2 options, HP might not support these options in a future PVU. If you are using a version 300 (or later) HP COBOL compiler to generate version 1 or version 2 data structures, use the INCLUDE STRUCTURES directive with the VERSION 1 or VERSION 2 option and remove the RELEASE1 or RELEASE2 option from the INCLUDE SQLDA directive.

`CPRULES`

is a keyword that is required if you specify a collation buffer.

collation-buffer

is a host variable that is a COBOL record name of the collation buffer. The DESCRIBE statement includes the COLLATIONS INTO clause, which directs SQL/MP to return collations to *collation-buffer*.

For more information, see Calculating the Lengths of the Names and Collation Buffers on page 10-20.

max-collation-size

is the maximum number of bytes you expect for any one collation.

Considerations

These guidelines apply to the use of the INCLUDE SQLDA directive:

- You must supply both the number of parameters you expect as well as the maximum lengths of their names. Be sure to specify sufficient numbers and sizes. You can specify large numbers to ensure that any data you might obtain at run time will fit. If you are not concerned about memory use, this approach is most efficient.
- Declare buffers in working storage. Use redefines to store differing data types.
- One reason to specify a collation buffer is to access collation rules in your COBOL code so that you can make comparisons using the same rules that SQL/MP does for a given column.

Examples

In this example, the INCLUDE SQLDA directive generates a version 300 SQLDA structure named SQLDAX with these buffers:

- NAMES-BUFFER, which reserves space for 20 names with a maximum length of 30 bytes each

- COLLATION-BUFFER, which supports collations with a maximum length of 512 bytes each

The INCLUDE STRUCTURES and INCLUDE SQLDA directives in the COBOL program are:

```
DATA DIVISION.
EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 315 END-EXEC.
...
EXEC SQL INCLUDE SQLDA (SQLDAX, 20, NAMES-BUFFER, 30,
                        CPRULES COLLATION-BUFFER, 512)
END-EXEC.
```

The HP COBOL compiler generates this SQLDA structure:

```
01 SQLDAX.
  05 EYE-CATCHER     PIC X(2)  VALUE "D1".
  05 NUM-ENTRIES     PIC S9(4) COMP VALUE 20.
  05 SQLVAR          OCCURS 20 TIMES.
    10 DATA-TYPE       PIC S9(4)  COMP VALUE 0.
    10 DATA-LEN        NATIVE-2.
    10 PRECISION       PIC S9(4)  COMP VALUE 0.
    10 NULL-INFO       PIC S9(4)  COMP VALUE 0.
    10 VAR-PTR         PIC S9(9)  COMP VALUE -999999.
    10 IND-PTR         PIC S9(9)  COMP VALUE -999999.
    10 CPRL-PTR        PIC S9(9)  COMP VALUE -999999.
    10 RESERVED        PIC S9(9) COMP VALUE -1.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 NAMES-BUFFER        PIC X(840).
01 COLLATION-BUFFER    PIC X(10320).
EXEC SQL END DECLARE SECTION END-EXEC.
```

The HP COBOL compiler generates a Declare Section for the NAMES-BUFFER and COLLATION-BUFFER declarations and determines their lengths:

```
NAMES-BUFFER = ((30 + 11) + 1) * 20
             =  840 bytes
COLLATION-BUFFER = (512 + 4) * 20
                 =  10320 bytes
```

Defining Buffers for Parameter and Variable Storage

Define storage for parameter values according to data type. For example:

```
* Define storage for all possible data types.  The VAR-PTR
* field in the input SQLDA points to this storage.
  01 PARAM-REC.
     02 PARAMS OCCURS 20 TIMES.
       03 P                      PIC  X(60).
       03 PCHAR    REDEFINES P   PIC  X(60).
       03 PVARCHAR REDEFINES P.
          04 LEN                 PIC S9(4) COMP.
          04 VAL                 PIC  X(58).
       03 PNUMERIC  REDEFINES P  PIC S9(15)V9(3) COMP.
       03 PINT      REDEFINES P  PIC  9(9) COMP.
```

```
        03 PDECIMAL  REDEFINES P  PIC S9(9)V9(3)  DISPLAY.
        03 PLARGINT  REDEFINES P  PIC  9(18) COMP.
        03 PSMLINT   REDEFINES P  PIC  9(4)  COMP.
```

Declare buffers in working storage. Use redefines to store differing data types.

Preparing the SQL Statement

Before preparing the statement, specify WHENEVER directives for error handling in the Procedure Division:

```
EXEC SQL
  WHENEVER SQLERROR PERFORM :100-HANDLE-ERROR END-EXEC.
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
```

If you are obtaining the SQL statement from a user, read the statement. In the Procedure Division, prepare the SQL statement. This example shows the sequence you can follow for the PREPARE statement, which assigns the name S1 to the SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
   01 STATEMENT-BUFFER  PIC X(256).
  ...
 EXEC SQL END DECLARE SECTION  END-EXEC.
  ...
 DISPLAY "Enter a new SQL statement:"
 ACCEPT STATEMENT-BUFFER.
  ...
* Check the contents of the statement (The sample program
* checks for permutations of "END" or "SAME")
 ...
EXEC SQL PREPARE S1 FROM :STATEMENT-BUFFER  END-EXEC.
```

Checking for Parameters

After you have prepared the statement, you can check to see if there are any input or output parameters (unless you know there are not any). Table 10-4 lists the information that the PREPARE statement stores in the SQLSA structure.

Table 10-4. SQLSA Contents after a Prepare Operation (page 1 of 2)

SQLSA Field	Description
INPUT-NUM	Number of input parameters in the SQL statement. Use this information to decide how many parameter values to solicit from the user.
INPUT-NAMES-LEN	Length of the buffer required to contain the names of the input parameters.
OUTPUT-NUM	Number of output parameters in the statement. Use this information to decide how many column values to report.

Table 10-4. SQLSA Contents after a Prepare Operation (page 2 of 2)

SQLSA Field	Description
OUTPUT-NAMES-LEN	Length of the buffer required to contain the names of the output parameters.
SQL-STATEMENT-TYPE	Type of statement being prepared. Use this information to decide what type of statement was entered. SQL-STATEMENT-TYPE can have these values: 1 Cursor SELECT 2 INSERT 3 UPDATE 4 DELETE 5 DDL statement 6 Run-time CONTROL TABLE 7 LOCK, UNLOCK, or FREE RESOURCES 8 GET VERSION OF *object*
OUTPUT-COLLATIONS-LEN	Length of the output collations buffer if the application uses collations.

To check the input parameters:

1. Retrieve the number of input parameters from INPUT-NUM OF SQLSA.
2. Specify a DESCRIBE INPUT statement to access input parameters:

```
EXEC SQL DESCRIBE INPUT S1
         INTO :IN-SQLDA NAMES INTO :IN-NAMESBUF
END-EXEC.
```

Checking for Output Parameters

To check for output parameters, you perform essentially the same set of operations described for input parameters except that the pointers point to the output SQLDA and names buffer and collations buffer. To get the descriptions of the output parameters into the output SQLDA, use the DESCRIBE statement instead of DESCRIBE INPUT.

To check for variables and allocate space if necessary, perform these steps:

1. Retrieve the number of output parameters from OUTPUT-NUM OF SQLSA.
2. Issue a DESCRIBE statement to access the output parameters. The DESCRIBE statement can include a COLLATIONS INTO clause, which directs SQL/MP to return collations to COLL-BUF:

```
EXEC SQL
  DESCRIBE S1
    INTO :OUT-SQLDA NAMES INTO :OUT-NAMESBUF
      COLLATIONS INTO :COLL-BUF
END-EXEC.
```

Calculating the Lengths of the Names and Collation Buffers

Use the names buffer to store the names of your input parameters (after a DESCRIBE INPUT operation) or the names of selected columns (after a DESCRIBE operation). SQL/MP returns a name to the names buffer as a VARCHAR item. A column name is qualified with the table name. The HP COBOL compiler processes the name as a group item defined as:

```
02  LEN      PIC 9(4) COMP.
02  VAL      PIC X(value-of-len).
```

A name with an odd number of characters is padded with a trailing blank to make the length an even number. The HP COBOL compiler determines the length in bytes of the names buffer:

```
names-buffer-length =
     ( EVEN ( max-name-length + 11 ) ) * sqlvar-count
```

The 11 bytes added to `name-string-size` are derived from the length field (2 bytes), table name (8 bytes), and period separator (1 byte), rounded to an even number.

The names buffer contains null strings for unnamed input parameters. For an output expression, the names buffer contains a null string.

The collation buffer receives copies of any collation objects used by columns in the query. SQL/MP returns a collation object to the collation buffer as a VARCHAR item. The HP COBOL compiler processes a collation as a group item defined as:

```
02  LEN      PIC 9(4) COMP.
02  VAL      PIC X(value-of-len).
```

The HP COBOL compiler determines the length in bytes of the collation buffer:

```
collation-buffer-length =
          ( max-collation-size + 4 ) * sqlvar-count
```

The 4 bytes added to `max-collation-size` is the length (LEN) field in the VARCHAR item.

Handling Parameters

If the SQL statement has parameters, the INPUT-NUM field (for input parameters) or OUTPUT-NUM field (for output parameters) of the SQLSA does not equal 0. If the statement has parameters, you must change the descriptions in the SQLDA to point to appropriate host variables. Change the corresponding data types and set up the SQLDA structure to point to the storage for variables referenced by the query. In addition, depending on your application, you must check and possibly set several SQLDA fields related to null values and scaled data.

Loop through the SQLVAR array in the input (for input parameters) or output (for output parameters) SQLDA structure. Loop *n* times, where *n* is the number of parameters from INPUT-NUM or OUTPUT-NUM, respectively. On each iteration, follow these steps:

1. Check the DATA-TYPE field. If necessary, adjust the data type and reset DATA-LEN and PRECISION accordingly. For an example of this, see Example 10-1 on page 10-25.
2. Allocate an amount of memory equal to DATA-LEN for the parameter.
3. Initialize EYE-CATCHER, NUM-ENTRIES, VAR-PTR, and IND-PTR in the SQLDA structure.
4. Use the NULL-INFO field to tell whether a parameter can contain a null value. Use the IND-PTR field to tell is the associated value is actually null.
5. To display column headings for output parameters (as SQLCI does), loop through the names buffer to read the corresponding name for each column and display the column names.

If you know the number and data types of your parameter values, you can set DATA-TYPE, DATA-LEN, and VAR-PTR.

The next subsections provide more information about each of these steps.

Checking the DATA-TYPE Field

The DATA-TYPE field in the SQLDA indicates the data type of the parameter. Check the DATA-TYPE field to decide how to store input parameter values and how to display output column values. When you evaluate data types, check for these numeric ranges:

0 – 63	CHARACTER data
64 – 127	VARCHAR data
128 – 191	VARCHAR data
192 – 212	Date-time or INTERVAL data

You might need to change the value of the DATA-TYPE field if the data type of the parameter you declared is compatible with the data type SQL uses.

Checking the DATA-LEN Field

If your program must handle numeric values with scale, you need to read scale information from the output SQLDA. DESCRIBE places this information in bits 0 through 7 of the DATA-LEN field in the SQLVAR entry.

If you can ignore scale, you can set bits 0 through 7 of the DATA-LEN field to 0, causing data truncation. Otherwise, COBOL data types handle scale. To determine whether scale is present, check whether DATA-LEN is greater than 255. If it is, the upper byte has a nonzero value, and scale information is present.

The same considerations apply if your program must handle precision for date-time, INTERVAL, FLOAT, or binary numeric values. The precision information is in the PRECISION field of the SQLVAR entry.

This example checks for scale in an output value and sets the scale information to 3 for numeric and decimal items. You would follow a similar procedure to handle a scaled input parameter value, using the input SQLDA instead of the output SQLDA.

```
* Set DATA-LEN to (3,8) for numeric and to (3,12)
* for decimal if scale exists.  DATA-TYPE between 127 and
* 150 is numeric:
  IF DATA-TYPE OF SQLVAR OF OUT-SQLDA (INDEX) > 127
  AND DATA-LEN OF SQLVAR OF OUT-SQLDA (INDEX) > 255
        IF DATA-TYPE OF SQLVAR OF OUT-SQLDA (INDEX) < 150

* Move a binary 3 to upper byte and a binary 8 to lower byte:
       MOVE 776 TO DATA-LEN OF SQLVAR OF OUT-SQLDA(INDEX)
       ELSE
* Move a binary 3 to upper byte and a binary 12 to lower
* byte:
       MOVE 780 TO DATA-LEN OF SQLVAR OF OUT-SQLDA (INDEX)
  END-IF
```

To find the decimal equivalents for binary values, start Inspect and enter the hexadecimal equivalent for the decimal value in each byte:

```
>INSPECT

* You are in INSPECT here
--display (%h0308) in d
776
--exit

* You are in your program here
... MOVE 776 TO DATA-LEN OF OUTPUT-SQLDA
```

Checking the PRECISION Field for Character Set ID

For character (including VARCHAR) parameters and variables, the PRECISION field contains the character set ID. When a dynamic SQL statement executes, SQL/MP checks the PRECISION field to ensure that the character set ID matches the expected character set of the parameter, which is determined by the COLUMNS.CHARACTERSET value.

If the character sets do not match, SQL/MP returns an error. If, however, the program expects an UNKNOWN character set and the CHARACTERSET value for the parameter indicates a single-byte character set, SQL/MP does not return an error.

Initializing the SQLDA Structure

When a program issues a DESCRIBE INPUT or DESCRIBE statement, the system supplies values for all fields of the SQLDA except EYE-CATCHER, NUM-ENTRIES, VAR-PTR, and IND-PTR. If an application supports parameters, it must explicitly reference the DATA-TYPE, DATA-LEN, and VAR-PTR fields in the SQLDA. In addition, this program must initialize these fields:

- EYE-CATCHER to point to the value D1 (for a version 2 or 300 SQLDA) or DA (for a version 1 SQLDA).
- VAR-PTR to point to the input or output data buffer. Because COBOL programs cannot generate addresses, you must call a TAL procedure, SQLADDR, to accomplish this task. SQLADDR takes the address of a host variable in Working Storage and places the address in VAR-PTR. To call SQLADDR, use the format in the next example. Its syntax is described in Section 5, SQL/MP System Procedures.
- IND-PTR to point to indicator variables, if any. If you are handling null values, check NULL-INFO. If NULL-INFO is 0, do not allocate any memory. If NULL-INFO is -1, allocate two bytes of memory for the indicator value and set IND-PTR to the address of the indicator variable.

  If the program does not process null values, set IND-PTR to an invalid address.

```
* For input data buffer (variable definitions appear in
* Example 10-1). INDEX is a loop counter--you are
* setting VAR-PTR for each input parameter:

  ENTER TAL "SQLADDR"
    USING P OF PARAMS(INDEX)
    GIVING VAR-PTR OF SQLVAR OF IN-SQLDA(INDEX)


* For output data buffer (variable definitions appear in
* Example 10-3). INDEX is a loop counter--you are
* setting VAR-PTR for each output parameter:

  ENTER TAL "SQLADDR"
    USING C OF COLUMN(INDEX)
    GIVING VAR-PTR OF SQLVAR OF OUT-SQLDA(INDEX)
```

Using the NULL-INFO and IND-PTR Fields

The DESCRIBE INPUT statement sets the NULL-INFO field depending on whether the prepared SQL statement includes a null indicator and not whether the parameter actually supports a null value. To determine if a parameter supports a null value, check the NULLALLOWED column in the COLUMNS table for the catalog where the table is registered.

The input and output SQLDA structures have two fields, NULL-INFO and IND-PTR, that are used for handling null values:

- NULL-INFO tells whether the input parameter or output variable can contain a null value, based on whether the prepared statement includes an associated null indicator parameter.

- IND-PTR points to a flag in Working-Storage that indicates whether the parameter actually was null. If the parameter or output variable is not null, you use the location referenced by VAR-PTR to indicate the value.

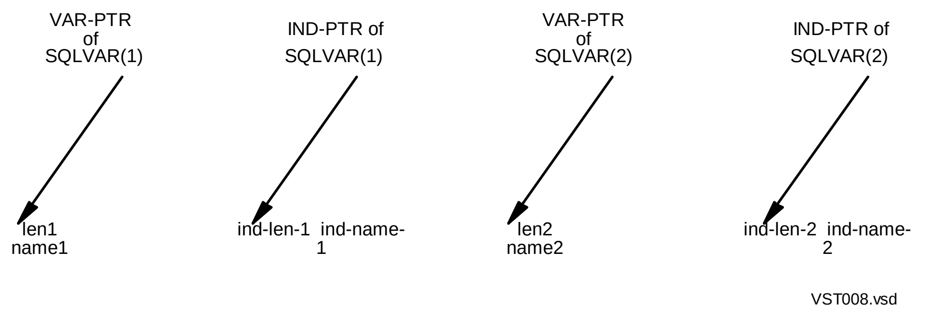
If your program processes indicator parameters, IND-PTR points to the indicator parameter associated with that input parameter in the names buffer after DESCRIBE INPUT executes. This behavior is parallel to that of VAR-PTR after DESCRIBE INPUT or DESCRIBE executes.

You access the IND-PTR in the SQLVAR array in the same way you access VAR-PTR: you call SQLADDR to make IND-PTR point to the Working-Storage flag that indicates whether the value is null. If you want all your parameters and output parameters to handle null values, your program should access IND-PTR every time it accesses VAR-PTR.

Figure 10-2 illustrates the structure of the names buffer immediately after DESCRIBE INPUT executes when indicator parameters are present for two parameters, where *`len`* is a 2-byte length, *`name`* is a parameter name, *`ind-len`* is the length of an indicator parameter name, and *`ind-name`* is an indicator parameter name.

Like parameter names, indicator variable names are blank padded to even lengths.

Figure 10-2. Names Buffer Structure



Prompting the User for Input Values

If there are input parameters, prompt the user for values before executing the SQL statement. Read the name of each parameter and prompt the user for each value in this way:

1. Loop through the names buffer and read each value into the data buffer you have allocated for the parameter, according to the data type of the value.
2. If the parameter can be null (NULL-INFO is -1) and the value entered was null, set the indicator variable at the location in IND-PTR to -1.

Before prompting for input, use the DATA-TYPE field in the SQLDA structure to evaluate input parameter values. Example 10-1 on page 10-25 shows one way to do this.

Example 10-1. Evaluating Input Parameter Values (page 1 of 2)

```
 DATA DIVISION.
* Loop counter:
  01 INDEX  PIC S9(4) COMP.

* Define storage for all possible data types.  The VAR-PTR
* field in the input SQLDA points to this storage.
  01 PARAM-REC.
     02 PARAMS OCCURS 20 TIMES.
       03 P                      PIC  X(60).
       03 PCHAR    REDEFINES P   PIC  X(60).
       03 PVARCHAR REDEFINES P.
          04 LEN                 PIC S9(4) COMP.
          04 VAL                 PIC  X(58).
       03 PNUMERIC  REDEFINES P  PIC S9(15)V9(3) COMP.
       03 PINT      REDEFINES P  PIC  9(9) COMP.
       03 PDECIMAL  REDEFINES P  PIC S9(9)V9(3)  DISPLAY.
       03 PLARGINT  REDEFINES P  PIC  9(18) COMP.
       03 PSMLINT   REDEFINES P  PIC  9(4)  COMP.

 PROCEDURE DIVISION.

 MOVE 1 TO INDEX
 ...
 PERFORM UNTIL INDEX IS > INPUT-NUM OF SQLSA
    ...
* Check for character data type:
* Guard against the case where the database column is longer
* than the data variable in Working-Storage.
   IF DATA-TYPE OF SQLVAR OF IN-SQLDA(INDEX) < 64
           ACCEPT PCHAR OF PARAMS(INDEX)
           IF DATA-LEN OF IN-SQLDA(INDEX) > 60
            MOVE 60 TO DATA-LEN OF SQLVAR OF IN-SQLDA(INDEX)
           END-IF
```

Example 10-1. Evaluating Input Parameter Values (page 2 of 2)

```
  ELSE

* Check for VARCHAR data type; store length and value in
* separate sub-fields:
    IF DATA-TYPE OF SQLVAR OF IN-SQLDA(INDEX) = 64
           ACCEPT VAL OF PVARCHAR OF PARAMS(INDEX)
           IF DATA-LEN OF IN-SQLDA(INDEX) > 58
            MOVE 58 TO DATA-LEN OF SQLVAR OF IN-SQLDA(INDEX)
           END-IF

* If you want SQL to check whether the string the user
* entered will fit into the database column, you can also
* determine the length of the user-supplied string and move
* that length to DATA-LEN OF SQLVAR OF IN-SQLDA(INDEX) and to
* LEN OF PVARCHAR OF PARAMS(INDEX).

    ELSE

* Check for 16-bit integer:
    IF DATA-TYPE OF SQLVAR OF IN-SQLDA(INDEX) <= 131
            ACCEPT PSMLINT OF PARAMS(INDEX)

   ELSE ...
* Continue to check for and store all possible data types.
```

Using the Names Buffer to Prompt for Input Parameter Values

You can use the names buffer to prompt the user for input parameter values, in which case the names buffer contains the names of the input parameters. You can also use the names buffer to display column names, in which case the names buffer contains the names of the columns.

The data returned to the names buffer is in this form:

len-1 *name*-1 *len*-2 *name*-2 ... *len*-*n* *name*-*n*

In this case, *name-1* represents the first parameter or column name, *name-2* the second, and *name-n* the last. The length information is a 2-byte integer (SQL data type PIC S9(4) COMP). All names with a length of an odd number of characters are padded with a blank to make the length an even number. When you display the names, you might want to check for this blank padding. Names for output expressions or unnamed input parameters appear as a null string with a length of 0.

To determine the names in the names buffer programmatically, you can write a routine to return the names structure when given the address of the column information desired. After the DESCRIBE INPUT or DESCRIBE statement executes, the VAR-PTR field of each SQLVAR entry in the input or output SQLDA contains the address of the length field associated with the name of the corresponding parameter or column in the names buffer.

You can use VAR-PTR to read the names from the names buffer only if you access the names buffer immediately following DESCRIBE INPUT or DESCRIBE. After you have set VAR-PTR to point to the data, you can no longer use VAR-PTR to access the names buffer and must loop through the names buffer to get the names.

Some examples of entries in the names buffer are:

Complete Entry	**Entry Part**	**Description**
`|04|ABCD|`	`|04|`	2-byte length 4-character string with value = 4
	`|ABCD|`	4-character string
`|06|ABCDE |`	`|06|`	2-byte length 4-character string with value = 6
	`|ABCDE |`	5-character string padded with 1 trailing blank
`|00||`	`|00|`	2-byte length with value = 0
	`||`	Null string

A complete names buffer with the names shown in this example might look like this:

```
|04|ABCD|06|ABCDE |00|
```

When reading the names buffer, check to see if NULL-INFO is -1. If so, read the length field for the indicator and add this length field to the index to skip to the next name in the names buffer.

Example 10-2 prompts for input.

Example 10-2. Prompting for Input

```
 DATA DIVISION.
 ...
  01 NAME          PIC X(30).
  01 NAME-IX       PIC S9(4) COMP.

* This variable will store the 2-byte length field:
  01 NAMESIZEX     PIC X(2).

* This variable redefines the 2-byte length field
* as an integer so you can perform arithmetic to advance
* through the buffer:
  01 NAMESIZE REDEFINES NAMESIZEX  PIC S9(4) COMP.

 PROCEDURE DIVISION.
 ...

  DISPLAY "ENTER PARAMETER VALUES: "
  MOVE 1 TO NAME-IX

* In PERFORM loop for a number of iterations equal to
* INPUT-NUM OF SQLSA:

* Store the 2-byte length field in variable NAMESIZEX:
  MOVE IN-NAMESBUF (NAME-IX : 1) TO NAMESIZEX (1 : 1)
  MOVE IN-NAMESBUF (NAME-IX + 1 : 1) TO NAMESIZEX (2 : 1)

* Move pointer NAME-IX past the LENGTH field and onto the
  name:
  COMPUTE NAME-IX = NAME-IX + 2

* If NAMESIZE > 0, display the name with a prompt character:
  MOVE SPACE TO NAME
  MOVE IN-NAMESBUF( NAME-IX : NAMESIZE ) TO NAME
  DISPLAY "?", NAME

* Position to the next length field:
  COMPUTE NAME-IX = NAME-IX + NAMESIZE

* ACCEPT the parameter value according to its data type:
          ...
```

Handling Null Values in Input Parameters

If your program handles null values on input, each parameter in the statement entered by the user or constructed by your program must have a corresponding indicator parameter to handle possible null values, or a run-time error will occur when a null value is encountered.

After DESCRIBE INPUT executes and for each input parameter described in an SQLVAR array in the input SQLDA, SQL/MP sets NULL-INFO to -1 if the input parameter in the prepared statement could have a null value (that is, if the prepared statement included a null indicator parameter).

If the user specifies a null value for the parameter, set the Working-Storage location referenced by IND-PTR to -1. SQL/MP checks this value and assumes a null value for the parameter.

If instead the user does not enter a null value for the input parameter, you can assign a 0 to the location pointed to by IND-PTR. SQL/MP checks IND-PTR, sees that IND-PTR indicates a nonnull value, and gets the parameter value from the Working-Storage location pointed to by VAR-PTR for the parameter value.

Note. The DESCRIBE INPUT statement sets the NULL-INFO field, depending on whether the prepared SQL statement includes a null indicator and not on whether the column in the table supports a null value. To determine if a column allows a null value, check the NULLALLOWED column in the COLUMN catalog table for the catalog where the table is registered.

Performing the Database Request

An application must process statements that are not SELECT statements differently than SELECT statements.

Processing NonSELECT Statements

To determine if a statement is not a SELECT statement, check SQL-STATEMENT-TYPE OF SQLSA to see if it equals 0. If so, perform these steps:

1. Begin a TMF transaction:

   ```
   EXEC SQL BEGIN WORK END-EXEC.
   ```

   (Depending on your transaction definition you might not want to do this for every statement.)

2. Execute a statement other than a SELECT statement.

   - If there were input parameters:

     ```
     EXEC SQL EXECUTE S1
       USING DESCRIPTOR :IN-SQLDA END-EXEC.
     ```

   - If there were no input parameters:

     ```
     EXEC SQL EXECUTE S1  END-EXEC.
     ```

3. End the TMF transaction (for both SELECT and other statements):

   ```
   EXEC SQL COMMIT WORK  END-EXEC.
   ```

   (Depending on your transaction definition you might not want to do this for every statement.)

Processing SELECT Statements

To determine if a statement is a SELECT statement, check SQL-STATEMENT-TYPE OF SQLSA to see if it equals 1. If so, perform these steps:

1. Declare a cursor to handle the SELECT statement:

   ```
   EXEC SQL DECLARE C1 CURSOR FOR S1  END-EXEC.
   ```

   For more information about dynamic cursors, see Using Dynamic SQL Cursors, following.

2. Begin a TMF transaction:

   ```
   EXEC SQL BEGIN WORK END-EXEC.
   ```

   (Depending on your transaction definition you might not want to do this for every statement.)

3. Open the cursor:

   ```
   EXEC SQL OPEN C1 USING DESCRIPTOR :IN-SQLDA  END-EXEC.
   ```

4. Execute a loop to fetch the values and display them:

   ```
   EXEC SQL FETCH C1 USING DESCRIPTOR :OUT-SQLDA  END-EXEC.
   * **SQLDA contains pointers to output data buffers
   ```

   Display the values in a format according to data type. (For a repetitive display of column names, use the output names buffer at this point and omit Steps 1through 3.)

   If you know in advance which columns to select, you could use this form of the FETCH statement:

   ```
     EXEC SQL FETCH cursor INTO :

   , :SAL  END-EXEC.
   * **Output parameters are :

    and :SAL
   ```

5. Close the cursor:

   ```
   EXEC SQL CLOSE C1  END-EXEC.
   ```

6. End the TMF transaction:

   ```
   EXEC SQL COMMIT WORK  END-EXEC.
   ```

   (Depending on your transaction definition you might not want to do this for every statement.)

Using Dynamic SQL Cursors

Dynamic SQL statements use cursors to process SELECT statements in the same way static SQL statements use cursors. The program reads rows from a table, one by one, and sends the column values to output data buffers specified in the program. This subsection describes some guidelines for the use of cursors. The order for executing statements for using a cursor with dynamic SQL operations is:

Operation	Description
PREPARE *statement-name* FROM :*host-variable*	Dynamically compiles the SELECT statement defining the cursor
Issue the DESCRIBE INPUT and DESCRIBE statements	
DECLARE *cursor-name* CURSOR FOR *statement-name*	Declares the cursor
OPEN *cursor-name* USING DESCRIPTOR *input-sqlda*	Opens the cursor and gets parameter values from an input data buffer in the program
Loop until end-of-file	
FETCH *cursor-name* USING DESCRIPTOR *output-sqlda*	Retrieves data and outputs column values to an output data buffer in the program
CLOSE *cursor-name*	Closes the cursor

Follow these guidelines when you declare and use a cursor:

- If you are using the HP COBOL or SQL compiler interface, you can use a host variable wherever you can use the *cursor-name* and *statement-name* parameters. For each new statement and cursor, store the name in the host variable before executing the statements.
- The DECLARE CURSOR, PREPARE, OPEN, FETCH, CLOSE, DELETE WHERE CURRENT, UPDATE WHERE CURRENT, DESCRIBE INPUT, and DESCRIBE statements for a particular cursor and its associated statement must all appear in the same procedure, unless you are using a foreign cursor. See Using Foreign Cursors on page 4-23.
- The PREPARE statement does not have to precede the other statements in the program listing order. However, the PREPARE statement must precede the DECLARE CURSOR statement and any DESCRIBE, DESCRIBE INPUT, OPEN, FETCH, and CLOSE statements (for extended dynamic SQL statements, where the cursor and statement names are stored in host variables). Foreign cursors do not have this restriction.

Using Cursors With a USING DESCRIPTOR Clause

If the program is handling input parameters with values entered at run time, use the USING DESCRIPTOR clause with the OPEN statement to specify values for the parameter values in the SELECT statement. The input SQLDA describes the input location for each parameter. The DESCRIBE INPUT statement fills in the SQLVAR entries in the SQLDA, and your program sets the VAR-PTR fields and prompts the user for values for the parameters.

You also use the USING DESCRIPTOR clause with the FETCH statement to write column values to an output buffer specified in the program's variable declarations. The output SQLDA describes a list of memory locations into which FETCH copies the data.

Using Cursors With an UPDATE WHERE CURRENT Clause

To use UPDATE WHERE CURRENT with a static SQL cursor, specify a FOR UPDATE OF clause with a column list in the DECLARE CURSOR statement. In contrast, to use UPDATE WHERE CURRENT with a dynamic SQL cursor, you must specify a FOR UPDATE OF clause in the SELECT statement that defines the cursor.

This example shows an UPDATE WHERE CURRENT operation with a dynamic SQL cursor. In the example, the host variable HOSTVAR contains the SELECT statement to define the cursor. The host variable :SALVAR receives the selected values.

```
PROCEDURE DIVISION.

  MOVE "SELECT SALARY FROM =EMPLOYEE FOR UPDATE OF SALARY"
   TO HOSTVAR.

  EXEC SQL PREPARE S1 FROM :HOSTVAR  END-EXEC.

  EXEC SQL DECLARE C1 CURSOR FOR S1  END-EXEC.

  EXEC SQL OPEN C1  END-EXEC.

  EXEC SQL FETCH C1 INTO :SALVAR  END-EXEC.

  EXEC SQL
    UPDATE =EMPLOYEE SET SALARY = SALARY * 1.20
      WHERE CURRENT OF C1
  END-EXEC.
```

Displaying Output

There are several ways to display column values. Three ways to display column values are:

- Using the Names Buffer to Display Output
- Displaying Names as Headings on page 10-33
- Using DATA-TYPE to Evaluate Column Values on page 10-33

Using the Names Buffer to Display Output

After the FETCH operation for each row, you can display output by performing essentially the same operations you did to prompt for input. However, you might display the column names at a different point.

For input, this sequence was described in Prompting the User for Input Values on page 10-24:

1. Get the length of the parameter name.
2. Advance to the name.
3. Display the name and ask for a value for that name.
4. Interpret the value entered according to data type.

To display output, one possible sequence is:

1. Get the length of the column name.
2. Advance to the name.
3. Interpret the data type of the column value.
4. Display the name with the value.

This sequence displays names and values repetitively. For example:

```
EMPNUM   2000
EMPNAME  MARILYN ROBERTS

EMPNUM   1566
EMPNAME  CATHERINE WILLIAMS

EMPNUM   1890
EMPNAME  RICHARD JONES
```

Displaying Names as Headings

Another possible sequence for displaying output would show the column names as headings, the way SQLCI does. To do this, loop (OUTPUT-NUM OF SQLSA) times:

- Get the length of the column name.
- Advance to the name.
- Display the name with some blank space.
- Advance to the next length field.

If you use this second method, you must execute a second loop to interpret and display the values, including enough blank space for each value to fall under its column heading.

Using DATA-TYPE to Evaluate Column Values

You can use DATA-TYPE to evaluate SELECT column values before displaying values. The code shown in Example 10-3 on page 10-34 illustrates one way to do this. This code displays the column names repetitively rather than displaying all the names as headings as SQLCI does.

Example 10-3. Displaying Output Column Values

```
 DATA DIVISION.

* Define storage for all possible data types for output columns:
 01 COLUMNS-REC.
    02 COLUMN OCCURS 20 TIMES.
       03 C                     PIC X(60).
       03 RCHAR    REDEFINES C  PIC X(60).
       03 RINT     REDEFINES C  PIC 9(9) COMP.
       03 RVARCHAR REDEFINES C.
          04 LEN                PIC S9(4) COMP.
          04 VAL                PIC X(30) .
       03 RNUMERIC REDEFINES C  PIC S9(15)V9(3) COMP.
       03 RDECIMAL REDEFINES C  PIC S9(9)V9(3)  DISPLAY.
       03 RLARGINT REDEFINES C  PIC  9(18) COMP.
       03 RSMLINT  REDEFINES C  PIC  9(4)  COMP.

* Define loop counter:
 01 INDEX   PIC S9(4) COMP.

* Define a variable to save the length of a column:
 01 CLEN    PIC S9(4) COMP.

* Define a variable to store the column name:
 01 NAME    PIC X(30).

* Get the column name from the names buffer and store in
* NAME.

 PROCEDURE DIVISION.

 PERFORM UNTIL INDEX IS > OUTPUT-NUM OF SQLSA
 ...
* Check for character data type:
 IF DATA-TYPE OF SQLVAR OF OUT-SQLDA(INDEX) < 64
       MOVE DATA-LEN OF SQLVAR OF OUT-SQLDA(INDEX) TO CLEN
       DISPLAY NAME, " = ",
               RCHAR OF COLUMN(INDEX) (1 : CLEN)
  ELSE
* Check for VARCHAR data type:
  IF DATA-TYPE OF SQLVAR OF OUT-SQLDA ( INDEX ) = 64
        MOVE LEN OF RVARCHAR OF COLUMN( INDEX ) TO CLEN
        DISPLAY NAME, " = ",
              VAL OF RVARCHAR OF COLUMN( INDEX ) ( 1 : CLEN )

  ELSE
* Check for 16-bit integer data type:
  IF DATA-TYPE OF SQLVAR OF OUT-SQLDA ( INDEX ) <= 131
         DISPLAY NAME , " = ", RSMLINT OF COLUMN( INDEX )

  ELSE ...
* Continue checking all possible data types.
```

Handling Null Results

If the value returned is null, SQL/MP checks NULL-INFO and moves a -1 into the location pointed to by IND-PTR. (Errors are returned if the value is null but NULL-INFO is 0 or if IND-PTR is an invalid address.)

Your program must check NULL-INFO to determine whether the value returned could be null. Handle null values as follows:

- If NULL-INFO is -1, check the indicator variable pointed to by IND-PTR of SQLVAR. If the indicator variable is also -1, then a null value was returned. Display something representing a null value (perhaps blanks or zeros). Otherwise, display the value in the location pointed to by VAR-PTR.
- If NULL-INFO is 0, display the value pointed to by VAR-PTR.

If ignoring null values, simply display the value pointed to by VAR-PTR.

Note. The DESCRIBE INPUT statement sets the NULL-INFO field, depending on whether the prepared SQL statement includes a null indicator and not on whether the column in the table supports a null value. To determine if a column supports a null value, check the NULLALLOWED column in the COLUMN catalog table for the catalog where the table is registered.

Constructing a Server that Interfaces With Pathway

A dynamic SQL application can accept statements directly from a screen interface like Pathway. The dynamic SQL statement (or part of a statement) accepted directly from the user is compiled and executed, and the program replies to the user.

In many cases, you can use Pathmaker to code applications for a Pathway environment. For detailed information on coding servers using SQL with Pathway screens, see the *Pathmaker Programming Guide*. This subsection presents guidelines for writing a dynamic SQL COBOL server that interfaces with Pathway. For general guidelines on writing COBOL servers for Pathway, see the *COBOL85 for NonStop Systems Manual* and the *Pathway/TS SCREEN COBOL Reference Manual*.

Constructing an SQL Statement from User Input

If the program functions with Pathway input, the user enters statements in a Pathway screen, and the SCREEN COBOL requester program sends the data to your program. Your program then performs the database request and replies to the SCREEN COBOL requester, which replies to the user through the Pathway screen.

If the user enters field values (instead of SQL statements), your program must construct the SQL statement from the input values. To construct the statement, first check values passed from the requester in the buffer to decide what the statement includes. As each value is read, you concatenate the corresponding text to form the statement.

Example

Suppose that an application screen describes a personnel record. If any column does not have a value, the user can enter N. LIST-MSG is the name of the request message you defined. The code in these examples checks the EMPNUM field in LIST-MSG and, if required, concatenates the text “EMPNUM” to the statement you are constructing:

One way to concatenate text in the statement is to use the STRING verb:

```
01 STATEMENT PIC X(256) VALUE SPACES.
...
MOVE "SELECT" TO STATEMENT.
...
IF EMPNUM OF LIST-MSG NOT = "N"
  STRING "EMPNUM" DELIMITED BY SIZE
    INTO STATEMENT
  END-STRING
ELSE IF EMPNAME OF LIST-MSG NOT = "N"
  ...
ELSE ...
```

Another way to concatenate text is to use PERFORM VARYING:

```
01 STATEMENT PIC X(256) VALUE SPACES.
01 INDX  PIC 999 COMP.
...
MOVE "SELECT" TO STATEMENT.
...
IF EMPNUM OF LIST-MSG NOT = "N"
    PERFORM VARYING INDX FROM 256 BY -1
    UNTIL STATEMENT (INDX:1) NOT = SPACE
    END-PERFORM
    MOVE "EMPNUM" TO STATEMENT(INDX + 2:)
ELSE IF EMPNAME OF LIST-MSG NOT = "N"
  ...
ELSE ...
```

In either of these examples, the statement could now contain the string SELECT EMPNUM. You continue to construct the entire statement based on the values the user entered.

Constructing a Reply Message

When you construct the reply message after the database request has been processed, you assign the output values to the reply message instead of formatting and displaying the values. You must define a reply message for every possible set of columns in the reply. The first field in the reply message record must contain the reply code to communicate with the SCREEN COBOL requester.

Sample Dynamic SQL Program

Example 10-4 shows an HP COBOL program that constructs a set of SELECT statements, prepares them, and retrieves the associated data.

Example 10-4. Sample Dynamic SQL Program (page 1 of 2)

```
?SQL
?INSPECT
?SYMBOLS
 IDENTIFICATION DIVISION.
 PROGRAM-ID.
       COBEXT.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  TANDEM/16.
 OBJECT-COMPUTER.  TANDEM/16.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.

   01 CURSORS.
      02 CURSOR-NAME PIC X(2) OCCURS 3 TIMES.
   01 STATEMENTS.
      02 STMT-NAME PIC X(2) OCCURS 3 TIMES.
   01 STMT-TEXT.
      02 TEXT-STRING PIC X(80) VALUE SPACES OCCURS 3 TIMES.
   01 IDX  PIC S9(4) COMP.
   01 DISPLAY-VALUE PIC Z(5)9.

  EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 ANSWER   PIC 9(6) COMP.

* Note: the following three host variables are declared here
* because COBOL does not allow an OCCURS clause in a host
* variable. Table entries from outside the Declare Section
* are therefore moved into these host variables one by one,
* and the SQL operations are performed on the host variables.

   01 TEMP-CURSOR-NAME PIC X(2).
   01 TEMP-STMT-NAME  PIC X(2).
   01 TEMP-STMT-TEXT PIC X(80) VALUE SPACES.
  EXEC SQL END DECLARE SECTION END-EXEC.

* Include SQLCA for error checking:
  EXEC SQL INCLUDE SQLCA END-EXEC.
* Declare Extended Storage Section for SQLIN structures:
?NOLIST
  EXTENDED-STORAGE SECTION.
?LIST
```

Example 10-4. Sample Dynamic SQL Program (page 2 of 2)

```
 PROCEDURE DIVISION.
  1000-DRIVER.
     PERFORM 3000-SPECIFY-ERROR-HANDLING.
     PERFORM 3100-PROCESS-QUERIES.
     STOP RUN.
  3000-SPECIFY-ERROR-HANDLING.
      EXEC SQL
        WHENEVER SQLERROR PERFORM :6000-HANDLE-ERROR
      END-EXEC.
  3100-PROCESS-QUERIES.
     MOVE "SELECT EMPNUM FROM =EMPLOYEE WHERE
-       " SALARY > 100000" TO TEXT-STRING OF STMT-TEXT (1)
     MOVE "SELECT EMPNUM FROM =EMPLOYEE WHERE
-       " SALARY < 20000" TO TEXT-STRING OF STMT-TEXT (2)
     MOVE "SELECT SALARY FROM =EMPLOYEE WHERE
-       " JOBCODE = 400" TO TEXT-STRING OF STMT-TEXT (3)

     MOVE "C1" TO CURSOR-NAME OF CURSORS (1)
     MOVE "C2" TO CURSOR-NAME OF CURSORS (2)
     MOVE "C3" TO CURSOR-NAME OF CURSORS (3)
     MOVE "S1" TO STMT-NAME OF STATEMENTS (1)
     MOVE "S2" TO STMT-NAME OF STATEMENTS (2)
     MOVE "S3" TO STMT-NAME OF STATEMENTS (3)

     PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > 3
      EXEC SQL BEGIN WORK END-EXEC
       MOVE TEXT-STRING OF STMT-TEXT (IDX) TO TEMP-STMT-TEXT
       MOVE STMT-NAME OF STATEMENTS (IDX) TO TEMP-STMT-NAME
       MOVE TEXT-STRING OF STMT-TEXT (IDX) TO TEMP-STMT-TEXT
       EXEC SQL PREPARE :TEMP-STMT-NAME FROM :TEMP-STMT-TEXT
       END-EXEC
       MOVE CURSOR-NAME OF CURSORS (IDX) TO TEMP-CURSOR-NAME
       EXEC SQL DECLARE :TEMP-CURSOR-NAME CURSOR FOR
         :TEMP-STMT-NAME
      END-EXEC
 EXEC SQL OPEN :TEMP-CURSOR-NAME END-EXEC
       PERFORM UNTIL SQLCODE < 0 OR SQLCODE = 100
         EXEC SQL FETCH :TEMP-CURSOR-NAME INTO
           :ANSWER END-EXEC
         IF SQLCODE >= 0 AND SQLCODE NOT = 100 THEN
           MOVE ANSWER TO DISPLAY-VALUE
           DISPLAY "ANSWER IS " DISPLAY-VALUE
         END-IF
       END-PERFORM
       EXEC SQL CLOSE :TEMP-CURSOR-NAME END-EXEC
      EXEC SQL COMMIT WORK END-EXEC
     END-PERFORM.
 6000-HANDLE-ERROR.
    ENTER TAL "SQLCADISPLAY" USING SQLCA
    STOP RUN.
```

11 Character Processing Rules (CPRL) Procedures

This section describes CPRL procedures that a COBOL program can call to process these collation objects:

- SQL collation—An SQL/MP object with file code 941 generated by the CREATE COLLATION statement
- Collation object—A Guardian file with file code 199 generated by the NLCP collation compiler ($system.system.NLCPCOMP)

Table 11-1 summarizes the CPRL procedures. These procedures are alphabetically described in detail on subsequent pages in this section.

Table 11-1. Character Processing Rules (CPRL) Procedures (page 1 of 2)

Procedure	Description
CPRL_ARE_	Determines if all characters in a string are in the character class defined by the specified SQL collation or collation object
CPRL_AREALPHAS_	Determines if all characters in a string are in the ALPHAS character class according to the specified SQL collation or collation object
CPRL_ARENUMERICS_	Determines if all characters in a string are numeric according to the specified SQL collation or collation object
CPRL_COMPARE1ENCODED_	Compares two strings (one encoded) according to the collation defined by an SQL collation or collation object
CPRL_COMPARE_	Compares two strings (neither encoded) according to the collation defined by an SQL collation or collation object
CPRL_COMPAREOBJECTS_	Compares two SQL collations or collation objects
CPRL_DECODE_	Decodes a string that has been encoded by CPRL_ENCODE_
CPRL_DOWNSHIFT_	Downshifts a character string according to the downshift rules in the specified SQL collation or collation object
CPRL_ENCODE_	Encodes a character string for comparison purposes
CPRL_GETALPHATABLE_	Extracts ALPHAS character class information from an SQL collation or collation object

Table 11-1. Character Processing Rules (CPRL) Procedures (page 2 of 2)

Procedure	Description
CPRL_GETCHARCLASSTABLE_	Extracts character class information from an SQL collation or collation object
CPRL_GETDOWNSHIFTTABLE_	Extracts downshift information from an SQL collation or collation object
CPRL_GETFIRST_	Finds the first string of a specified length according to an SQL collation or collation object
CPRL_GETLAST_	Finds the last string of a specified length according to an SQL collation or collation object
CPRL_GETNEXTINSEQUENCE_	Finds the next string after a specified string according to an SQL collation or collation object
CPRL_GETNUMTABLE_	Extracts numeric character class information from an SQL collation or collation object
CPRL_GETSPECIALTABLE_	Extracts SPECIALS character class information from an SQL collation or collation object
CPRL_GETUPSHIFTTABLE_	Extracts an array that can be used for upshifting
CPRL_INFO_	Returns information about a collation contained in an SQL collation or collation object
CPRL_READOBJECT_	Reads an collation object (with file code 199) from a Guardian file into a buffer
CPRL_UPSHIFT_	Upshifts a character string according to the upshift rules in the specified SQL collation or collation object

COBOLEXT File

To call the CPRL procedures, which are written in TAL, use the COBOL ENTER TAL statement. The COBOLEXT file contains source declarations for these procedures (as well as for other system procedures). You might need to check with your system administrator to make sure the COBOLEXT file for the procedures you use in your program are available on your system. For more information about the COBOLEXT file and the ENTER TAL statement, see the *COBOL85 for NonStop Systems Manual*.

CPRL Error Codes

Each CPRL procedure returns specific error codes, which are listed in each procedure description. An error code of zero (0) indicates that the operation was successful.
All other CPRL error codes are less than zero, so they can be distinguished from file-system errors, which are always positive. The condition code (CC) setting has no meaning after the execution of a CPRL procedure.

CPRL_ARE_

The CPRL_ARE_ procedure determines if all characters in a string are in the character class defined by the specified CPRL. You can also call CPRL_ARE_ to scan a string for the first character not in a specific character class.

```
ENTER TAL "CPRL_ARE_" USING
                      classname,
                      classnamelength,
                      inputstring,
                      inputstringlength,
                      exceptcharaddr,
                      cprladdr
                      GIVING errorcode.
```

The CPRL_ARE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
–4	The version of the SQL collation or collation object is not supported.
–5	The user-specified character class does not exist in the specified SQL collation or collation object.
–6	The input string contains a character not in the specified character class.

classname pic X(*classnamelength*) input

is an array containing the name of the specified character class.

classnamelength pic S9(4) input

is the number of bytes in the character class name *classname*.

inputstring pic X(*inputstringlength*) input

is a string containing the data to be scanned.

inputstringlength pic S9(4) input

is the number of bytes to be scanned in *inputstring*.

exceptcharaddr pic S9(9) output

is set as follows:

- If the call is successful, all the scanned characters are in the character class defined by the specified SQL collation or collation object, and *exceptcharaddr* is set as follows:

  exceptcharaddr = address(*inputstring*) + *inputstringlength*

- If -6 is returned, the first character in *istring* not in the specified character class was found; *exceptcharaddr* is set to the address of this character.
- For other error codes, *exceptcharaddr* is set to an invalid address.

cprladdr `pic S9(9)` input

is a pointer to the SQL collation or collation object.

CPRL_AREALPHAS_

The CPRL_AREALPHAS_ procedure determines if all characters in a string are in the ALPHAS character class according to a specified SQL collation or collation object. You can also use this procedure to scan for the first character in the string that is not in the ALPHAS character class.

```
ENTER TAL "CPRL_AREALPHAS_" USING
                            inputstring,
                            inputstringlength,
                            exceptcharaddr,
                            cprladdr
                            GIVING errorcode.
```

The CPRL_AREALPHAS_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-6	The input string contains a character not in the ALPHAS character class.

inputstring `pic X(`*inputstringlength*`)` input

is an array containing the string to be scanned.

inputstringlength `pic S9(4)` input

is the number of bytes to be scanned in *inputstring*.

exceptcharaddr `pic S9(9)` output

is set as follows:

- If the call is successful, all the scanned characters are ALPHAS character class, and *exceptcharaddr* is set:

  exceptcharaddr = address(*inputstring*) + *inputstringlength*

- If –6 is returned, the first character in *inputstring* that is not in the ALPHAS character class was found; *exceptcharaddr* is set to the address of this character.

- For other error codes, *exceptcharaddr* is set to an invalid address.

```
cprladdr                  pic S9(9)                       input
```

is a pointer to the SQL collation or collation object.

CPRL_ARENUMERICS_

The CPRL_ARENUMERICS_ procedure determines if all characters in a string are numeric according to the specified SQL collation or collation object. You can also use CPRL_ARENUMERICS_ to scan for the first nonnumeric character in a string.

```
ENTER TAL "CPRL_ARENUMERICS_" USING
                              inputstring,
                              inputstringlength,
                              exceptcharaddr,
                              cprladdr,
                              GIVING errorcode.
```

The CPRL_ARENUMERICS_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-6	The input string contains a character not in the specified character class.

```
inputstring               pic X(inputstringlength)        input
```

is an array containing the data to be scanned.

```
inputstringlength         pic S9(4)                       input
```

is the number of bytes in *istring* to be scanned.

```
exceptcharaddr            pic S9(9)                       output
```

is set as follows:

- If the call is successful, all the scanned characters are numeric characters, and *exceptcharaddr* is set:

  exceptcharaddr = address(*inputstring*) + *inputstringlength*
- If -6 is returned, the first nonnumeric character in *inputstring* was found; *exceptcharaddr* is set to the address of this character.
- For other error codes, *exceptcharaddr* is set to an invalid address.

cprladdr `pic S9(9)` input

is a pointer to the SQL collation or collation object.

CPRL_COMPARE1ENCODED_

The CPRL_COMPARE1ENCODED_ procedure compares two strings according to an SQL collation or collation object. The first string is assumed to be in encoded form, and the second is assumed to be in original (not encoded) form. For strings of unequal length, the procedure logically pads the shorter string with blanks.

Use the CPRL_COMPARE1ENCODED_ procedure to compare a constant with a set of values in one pass. The procedure encodes as much of the second string as necessary to perform the compare, and the overhead of repeatedly encoding the constant is saved.

```
ENTER TAL "CPRL_COMPARE1ENCODED_" USING
                                  string1,
                                  string1length,
                                  string2,
                                  string2length,
                                  result,
                                  cprladdr
                                  GIVING errorcode.
```

The CPRL_COMPARE1ENCODED_ procedure returns these error codes:

Code	Description
0	The operation was successful.
–2	The SQL collation or collation object is invalid.
–4	The version of the SQL collation or collation object is not supported.

string1 `pic X(`*string1length*`)` input

is an array containing the first string to be compared. *string1* is assumed to be in encoded form.

string1length `pic S9(4)` input

is the number of bytes in *string1* to be compared.

string2 `pic X(`*string2length*`)` input

is an array containing the second string to be compared. *string2* is assumed to be in original (not encoded) form.

string2length `pic S9(4)` input

is the length of *string2*.

`result` pic S9(4) output

indicates the result of the comparison:

`-1`	The first operand is less than the second
`0`	The operands collate equally
`1`	The first operand is greater than the second

For error codes other than 0 (zero), *result* is meaningless.

`cprladdr` pic S9(9) input

is a pointer to the SQL collation or collation object.

CPRL_COMPARE_

The CPRL_COMPARE_ procedure compares two strings according to an SQL collation or collation object. Both strings are assumed to be in original (not encoded) form. For strings of unequal length, CPRL_COMPARE_ pads the shorter string with blanks.

Use CPRL_COMPARE_ for isolated compares. Only the necessary part of each string is encoded to perform the compare. However, if the same data is compared repeatedly, use the CPRL_ENCODE_ and CPRL_COMPARE1ENCODED_ procedures (or CPRL_ENCODE_ with binary compares).

```
ENTER TAL "CPRL_COMPARE_" USING
                          string1,
                          string1length,
                          string2,
                          string2length,
                          result,
                          cprladdr
                          GIVING errorcode.
```

The CPRL_COMPARE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

`string1` pic X(*string1length*) input

is an array containing the first string to be compared.

`string1length` pic S9(4) input

is the length in bytes of *string1*.

`string2` pic X(*string2length*) input

is an array containing the second string to be compared.

`string2length` pic S9(4) input

is the length in bytes of *string2*.

`result` pic S9(4) output

indicates the result of the comparison, if the error code is 0 (zero):

-1	*string1* is less than *string2*.
0	The strings collate equally.
1	*string1* is greater than *string2*.

`cprladdr` pic S9(9) input

is a pointer to the SQL collation or collation object.

CPRL_COMPAREOBJECTS_

The CPRL_COMPAREOBJECTS_ procedure compares two SQL collations or collation objects to determine whether they are equal.

```
ENTER TAL "CPRL_COMPAREOBJECTS_" USING
                                 cprladdr1,
                                 cprladdr2
                                 GIVING errorcode.
```

The CPRL_COMPAREOBJECTS_ procedure returns these error codes:

Code	Description
0	The operation was successful; the SQL collations or collation objects are equal.
−2	The SQL collation or collation object is invalid.
−4	The version of the SQL collation or collation object is not supported.
−21	The collations in the two specified SQL collations or collation objects do not match.

`cprladdr1` pic S9(9) input

is the address of the first SQL collation or collation object.

`cprladdr2` pic S9(9) input

is the address of the second SQL collation or collation object.

CPRL_DECODE_

The CPRL_DECODE_ procedure decodes a string that has been encoded by the CPRL_ENCODE_ procedure. If the same (or equivalent) SQL collation is used for both CPRL_ENCODE_ and CPRL_DECODE_, the decoded string will be equal to the original string with respect to that SQL collation.

Because encoding is not generally a one-to-one function, the decoded string might not be identical to the original string. For example, an SQL collation that is case-insensitive might produce a decoded string with different case letters than the original string. The string ABCDE might encode to a value, which is aBcDe when decoded.

```
ENTER TAL "CPRL_DECODE_" USING
                         encodedstring,
                         encodedstringlength,
                         decodedstring,
                         decodedstringmaxlength,
                         decodedstringlength,
                         cprladdr
                         GIVING errorcode.
```

The CPRL_DECODE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
–2	The SQL collation or collation object is invalid.
–4	The version of the SQL collation or collation object is not supported.
–20	The user-specified buffer is not large enough to receive the returned string.

encodedstring pic X(*encodedstringlength*) input

is an array containing the data to be decoded.

encodedstringlength pic S9(4) input

is the number of bytes in *encodedstring* to be decoded.

decodedstring pic X(*decodedstringmaxlength*) output

is an array in which CPRL_DECODE_ returns the decoded string. Overlapping *encodedstring* and *decodedstring* causes unpredictable results.

decodedstringmaxlength pic S9(4) input

specifies the maximum length of *decodedstring*.

```
decodedstringlength         pic S9(4)                        output
```

is the number of bytes of *encodedstring* that were decoded. CPRL_DECODE_ pads the remainder of *decodedstring* with blanks up to *decodedstringmaxlength*.

```
cprladdr                    pic S9(9)                        input
```

is a pointer to the SQL collation or collation object.

CPRL_DOWNSHIFT_

The CPRL_DOWNSHIFT_ procedure downshifts a character string according to the downshift rules in a specified SQL collation or collation object.

```
ENTER TAL "CPRL_DOWNSHIFT_" USING
                            inputstring,
                            inputstringlength,
                            shiftedstring,
                            shiftedstringmaxlength,
                            shiftedstringlength,
                            cprladdr
                            GIVING errorcode.
```

The CPRL_DOWNSHIFT_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-20	The user-specified buffer is not large enough to receive the returned string.
-21	The collations in the two specified SQL collations or collation objects do not match.

```
inputstring                 pic X(inputstringlength)         input
```

is an array in which CPRL_UPSHIFT_ returns the downshifted string.

```
inputstringlength           pic S9(4)                        input
```

is the number of bytes to be downshifted in *inputstring*.

```
shiftedstring               pic X(shiftedstringmaxlength)    output
```

is an array in which CPRL_DOWNSHIFT_ returns the downshifted string.

The values for *inputstring* and *shiftedstring* can be equal, but other values can cause unpredictable results.

shiftedstringmaxlength `pic S9(4)` `input`

specifies the maximum length of *shiftedstring*, which must be greater than equal to *inputstring*.

shiftedstringlength `pic S9(4)` `output`

specifies the length of the downshifted string returned in *shiftedstring*.

cprladdr `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

CPRL_ENCODE_

The CPRL_ENCODE_ procedure encodes a character string so that a subsequent binary comparison will produce the proper results for the specified SQL collation. Use CPRL_ENCODE_ in situations where the number of encodings required is substantially less than the number of comparisons (for example, during a sort).

```
ENTER TAL "CPRL_ENCODE_" USING
                         decodedstring,
                         decodedstringlength,
                         encodedstring,
                         encodedstringmaxlength,
                         encodedstringlength,
                         cprladdr
                         GIVING errorcode.
```

The CPRL_ENCODE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-20	The user-specified buffer is not large enough to receive the returned string.

decodedstring `pic X(`*decodedstringlength*`)` `input`

is an array containing data to be encoded.

decodedstringlength `pic S9(4)` `input`

is the number of bytes in *decodedstring* to be encoded.

encodedstring `pic X(`*encodedstringmaxlength*`)` `output`

is an array in which CPRL_ENCODE_ returns the encoded string. Overlapping *decodedstring* and *encodedstring* causes unpredictable results.

encodedstringmaxlength `pic S9(4)` `input`

specifies the maximum length of *encodedstring*.

encodedstringlength `pic S9(4)` `output`

is the number of bytes of *decodedstring* that were encoded. CPRL_ENCODE_ pads the remainder of *decodedstring* with encoded blanks up to *decodedstringmaxlength*.

cprladdr `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

CPRL_GETALPHATABLE_

The CPRL_GETALPHATABLE_ procedure extracts ALPHAS character class information for single-byte character sets from an SQL collation or collation object.

```
ENTER TAL "CPRL_GETALPHATABLE_" USING
                                array,
                                cprladdr
                                GIVING errorcode.
```

The CPRL_GETALPHATABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

array `pic X(256)` `output`

is a 256-byte array specified by the user. If the call is successful, CPRL_GETALPHATABLE_ sets each byte in *array* as follows:

1	The corresponding character code in the SQL collation or collation object is in the ALPHAS character class.
0	The corresponding character code in the SQL collation or collation object is not in the ALPHAS character class.

If the call is unsuccessful, *array* is not modified.

cprladdr `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

CPRL_GETCHARCLASSTABLE_

The CPRL_GETCHARCLASSTABLE_ procedure extracts character class information from an SQL collation or collation object for a user-specified character class.

```
ENTER TAL "CPRL_GETCHARCLASSTABLE_" USING
                                    array,
                                    cprladdr,
                                    classname,
                                    classnamelength,
                                    GIVING errorcode.
```

The CPRL_GETCHARCLASSTABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-5	The user-specified character class does not exist in the specified SQL collation or collation object.

`array`   `pic X(256)`   `output`

is a 256-byte array specified by the user. If the call is successful, CPRL_GETCHARCLASSTABLE_ sets each byte in *array* as follows:

1	The corresponding character code in the SQL collation or collation object is in the character class specified by *classname*.
0	The corresponding character code in the SQL collation or collation object is not in the specified character class.

If the call is unsuccessful, *array* is not modified.

`cprladdr`   `pic S9(9)`   `input`

is a pointer to the SQL collation or collation object.

`classname`   `pic X(classnamelength)`   `input`

is the name of the user-specified character class.

`classnamelength`   `pic S9(4)`   `input`

is the length of *classname* in bytes.

CPRL_GETDOWNSHIFTTABLE_

The CPRL_GETDOWNSHIFTTABLE_ procedure extracts downshift information from an SQL collation or collation object.

```
ENTER TAL "CPRL_GETDOWNSHIFTTABLE_" USING
                                    array,
                                    cprladdr
                                    GIVING errorcode.
```

The CPRL_GETDOWNSHIFTTABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

array pic X(256) output

is a 256-byte array specified by the user.

If the call is successful, CPRL_GETDOWNSHIFTTABLE_ sets each byte in *array* to the downshifted version of the corresponding character in the SQL collation or collation object.

If the call is unsuccessful, *array* is not modified.

cprladdr pic S9(9) input

is a pointer to the SQL collation or collation object.

CPRL_GETFIRST_

The CPRL_GETFIRST_ procedure finds the first string of a specified length according to an SQL collation or collation object.

This procedure replaces the practice of using a string of hexadecimal zeros to generate the first string of a specified length, which does not work correctly for non-binary collating sequences.

```
ENTER TAL "CPRL_GETFIRST_" USING
                           firststring,
                           firststringmaxlength,
                           firststringlength,
                           cprladdr
                           GIVING errorcode.
```

The CPRL_GETFIRST_ procedure returns these error codes:

Code	Description
`0`	The operation was successful.
`-2`	The SQL collation or collation object is invalid.
`-4`	The version of the SQL collation or collation object is not supported.

`firststring`	`pic X(`*`firststringmaxlength`*`)`	`output`

is an array in which CPRL_GETFIRST_ returns the first string.

`firststringmaxlength`	`pic S9(4)`	`input`

is the maximum length of *`firststring`*.

`firststringlength`	`pic S9(4)`	`output`

specifies the number of bytes of *`firststring`* that were scanned. If the call is successful, *`firststringmaxlength`* and *`firststringlength`* will be equal.

`cprladdr`	`pic S9(9)`	`input`

is a pointer to the SQL collation or collation object.

CPRL_GETLAST_

The CPRL_GETLAST_ procedure finds the last string of a specified length according to an SQL collation or collation object.

This procedure replaces the practice of using a string of binary ones to generate the last string of a specified length, which does not work correctly for non-binary collating sequences.

```
ENTER TAL "CPRL_GETLAST_" USING
                          laststring,
                          laststringmaxlength,
                          laststringlength,
                          cprladdr,
                          GIVING errorcode.
```

The CPRL_GETLAST_ procedure returns these error codes:

Code	Description
`0`	The operation was successful.
`-2`	The SQL collation or collation object is invalid.
`-4`	The version of the SQL collation or collation object is not supported.

`laststring`	`pic X(`*`laststringmaxlength`*`)`	`output`

is an array in which CPRL_GETFIRST_ returns the last string.

`laststringmaxlength` `pic S9(4)` `input`

specifies the maximum length of *`laststring`*.

`laststringlength` `pic S9(4)` `output`

specifies the number of bytes of *`laststring`* that were scanned. (When CPRL_GETFLAST_ is successful, *`laststringlength`* and *`laststringmaxlength`* will be equal.)

`cprladdr` `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

CPRL_GETNEXTINSEQUENCE_

The CPRL_GETNEXTINSEQUENCE_ procedure finds the next string after a specified string according to an SQL collation or collation object.

This procedure replaces the practice of adding 1 to the least significant character of a string to find the next greater string, which does not work correctly for non-binary collating sequences.

```
ENTER TAL "CPRL_GETNEXTINSEQUENCE_" USING
                                    inputstring,
                                    inputstringlength,
                                    nextstring,
                                    nextstringmaxlength,
                                    nextstringlength,
                                    cprladdr
                                    GIVING errorcode.
```

The CPRL_GETNEXTINSEQUENCE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
−2	The SQL collation or collation object is invalid.
−4	The version of the SQL collation or collation object is not supported.
−20	The user-specified buffer is not large enough to receive the returned string.
−23	The *`inputstring`* parameter is already the maximum string of length *`inputstringlength`*.
−24	The input string is longer than the maximum length (256 bytes).

`inputstring` `pic X(inputstringlength)` `input`

is an array containing the input string.

`inputstringlength` `pic S9(4)` `input`

is the number of bytes in the input string *`inputstring`*.

nextstring `pic X(`*nextstringmaxlength*`)` `output`

is an array in which CPRL_GETNEXTINSEQUENCE_ returns the next string. Overlapping *inputstring* and *nextstring* causes unpredictable results.

nextstringmaxlength `pic S9(4)` `input`

specifies the maximum length of *nextstring*. The returned value is padded with blanks as necessary to fill *nextstring* for this length. In most cases, set *nextstring* to the same value as *inputstring*.

nextstringlength `pic S9(4)` `output`

specifies the number of bytes of *nextstring* that were scanned. (If CPRL_GETNEXTINSEQUENCE_ is successful, *nextstringlength* and *nextstringmaxlength* are equal.)

CPRL_GETNEXTINSEQUENCE_ pads *nextstring* with blanks up to *nextstringmaxlength*, and *nextstringlength* is the length of *nextstring* up to the point where the blank begin (*nextstringlength* should also be the same as *inputstringlength*).

cprladdr `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

CPRL_GETNUMTABLE_

The CPRL_GETNUMTABLE_ procedure extracts numeric character class information from an SQL collation or collation object.

```
ENTER TAL "CPRL_GETNUMTABLE_" USING
                              array,
                              cprladdr
                              GIVING errorcode.
```

The CPRL_GETNUMTABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

array `pic X(256)` output

is a 256-byte array specified by the user. If the call is successful, CPRL_GETNUMTABLE_ sets each byte in *array* as follows:

`1`	The corresponding character code in the SQL collation or collation object is numeric.
`0`	The corresponding character code in the SQL collation or collation object is not numeric.

If the call is unsuccessful, *array* is not modified.

cprladdr `pic S9(9)` input

is a pointer to the SQL collation or collation object.

CPRL_GETSPECIALTABLE_

The CPRL_GETSPECIALTABLE_ procedure extracts SPECIALS character class information from an SQL collation or collation object, if the SPECIALS character class exists.

If the SPECIALS character class does not exist, CPRL_GETSPECIALTABLE_ creates it. In this case, characters are considered SPECIALS if they are not ALPHAS or NUMERICS. (The ALPHAS and NUMERICS character classes exist in all SQL collations or collation objects.)

```
ENTER TAL "CPRL_GETSPECIALTABLE_" USING
                                  array,
                                  cprladdr
                                  GIVING errorcode.
```

The CPRL_GETSPECIALTABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

array `pic X(256)` output

is a 256-byte array specified by the user. If the call is successful, CPRL_GETSPECIALTABLE_ sets each byte in *array* as follows:

`1`	The corresponding character code in the SQL collation or collation object is in the SPECIALS character class.
`0`	The corresponding character code in the SQL collation or collation object is not in the SPECIALS character class.

If the call is unsuccessful, *array* is not modified.

```
cprladdr                   pic S9(9)                        input
```

is a pointer to the SQL collation or collation object.

CPRL_GETUPSHIFTTABLE_

The CPRL_GETUPSHIFTTABLE_ procedure extracts upshift information from an SQL collation or collation object.

```
ENTER TAL "CPRL_GETUPSHIFTTABLE_" USING
                                  array,
                                  cprladdr
                                  GIVING errorcode.
```

The CPRL_GETUPSHIFTTABLE_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.

```
array                      pic X(256)                       output
```

is a 256-byte array specified by the user.

If the call is successful, CPRL_GETUPSHIFTTABLE_ sets each byte in *array* to the upshifted version of the corresponding character code in the SQL collation or collation object.

If the call is unsuccessful, *array* is not modified.

```
cprladdr                   pic S9(9)                        input
```

is a pointer to the SQL collation or collation object.

CPRL_INFO_

The CPRL_INFO_ procedure returns information about an SQL collation or collation object. (The SQL CREATE COLLATION statement uses this procedure to determine the characteristics of SQL collations.)

```
ENTER TAL "CPRL_INFO_" USING
                       cprladdr
                    [ , cprlsize          ]
                    [ , is1to1            ]
                    [ , lengtheningfactor ]
                    [ , characterset      ]
                    [ , version           ]
                       GIVING errorcode.
```

The CPRL_INFO_ procedure returns these error codes:

Code	Description
0	The operation was successful.
-2	The SQL collation or collation object is invalid.
-4	The version of the SQL collation or collation object is not supported.
-20	The user-specified buffer is not large enough to receive the returned string.

`*cprladdr*` `pic S9(9)` `input`

is a pointer to the SQL collation or collation object.

`*cprlsize*` `pic S9(4)` `output`

is the length in bytes of the SQL collation or collation object.

`*is1to1*` `pic S9(4)` `output`

is set as follows:

1	The encoding for this SQL collation or collation object is a one-to-one map.
0	The encoding is not a one-to-one map.

`*lengtheningfactor*` `pic S9(4)` `output`

specifies the maximum lengthening that encodings can cause. (That is, for a specified string, the encoding is not more than *lengtheningfactor* times the original string length. For SQL collations or collation objects that preserve (or shorten) the length on encoding, *lengtheningfactor* is 1.)

`*characterset*` `pic S9(4)` `output`

specifies the character set assumed by the SQL collation or collation object:

0	UNKNOWN	105	ISO88595
101	ISO88591	106	ISO88596
102	ISO88592	107	ISO88597
103	ISO88593	108	ISO88598
104	ISO88594	109	ISO88599

`*version*` `pic S9(4)` `output`

is the format version of the SQL collation or collation object.

CPRL_READOBJECT_

The CPRL_READOBJECT_ procedure reads a collation object from a Guardian disk file (file code 199) into a user-specified buffer. CPRL_READOBJECT_ does not read SQL collations (file code 941) generated by a CREATE COLLATION statement.

```
ENTER TAL "CPRL_READOBJECT_" USING
                             buffer
                             bufferlength
                             objectlength
                             filename
                             filenamelength
                             cprladdr
                             GIVING errorcode.
```

The CPRL_READOBJECT_ procedure returns these error codes:

Code	Description
0	The operation was successful.
–2	The SQL collation or collation object is invalid.
–4	The version of the SQL collation or collation object is not supported.
–11	The user-specified buffer is too small for the SQL collation or collation object.
–12	The CPRL_READOBJECT_ local buffer is too small for the SQL collation or collation object.
–13	An error occurred during a call to the FNAMEEXPAND procedure for the Guardian file name.
–14	The file code of the Guardian file containing the collation object is not 199.

If a file-system error occurs, CPRL_READOBJECT_ return a file-system error code rather than a CPRL error code. File-system error codes are always positive, whereas CPRL error codes are less than or equal to zero (0).

buffer pic S9(4) output

is a user-supplied buffer to which CPRL_READOBJECT_ returns the collation object if the call is successful. CPRL_READOBJECT_ uses a local 4 KB buffer allocated on the data stack. If you are concerned about stack size limitations, use this procedure with caution.

bufferlength pic S9(4) input

is the size of *buffer* in bytes.

objectlength pic S9(4) output

is the actual length in bytes of the collation object read into *buffer*.

filename	pic X(*filenamelength*)	input

is the Guardian file name in external format containing the collation object. The file code for *filename* must be 199.

filenamelength	pic S9(4)	input

is the length in bytes of *filename*.

cprladdr	pic S9(9)	output

is the address of the collation object if 0 (zero) is returned. Otherwise, *cprladdr* is set to an invalid address.

CPRL_UPSHIFT_

The CPRL_UPSHIFT_ procedure upshifts a character string according to the upshift rules in the specified SQL collation or collation object.

```
ENTER TAL "CPRL_UPSHIFT_" USING
                          inputstring,
                          inputstringlength,
                          shiftedstring,
                          shiftedstringmaxlength,
                          shiftedstringlength,
                          cprladdr
                          GIVING errorcode.
```

The CPRL_UPSHIFT_ procedure returns these error codes:

Code	Description
0	The operation was successful.
–2	The SQL collation or collation object is invalid.
–4	The version of the SQL collation or collation object is not supported.
–20	The user-specified buffer is not large enough to receive the returned string.

inputstring	pic X(*inputstringlength*)	input

is an array containing the data to be upshifted.

inputstringlength	pic S9(4)	input

is the number of bytes in *inputstring* to be upshifted.

shiftedstring	pic X(*shiftedstringmaxlength*)	output

is an array in which CPRL_UPSHIFT_ returns the upshifted string.

The values for *inputstring* and *shiftedstring* can be equal, but other values can cause unpredictable results.

shiftedstringmaxlength	`pic S9(4)`	`input`

specifies the maximum length of *shiftedstring*, which must be greater than or equal to *inputstringlength*.

shiftedstringlength	`pic S9(4)`	`output`

specifies the length of the upshifted string returned in *shiftedstring*.

cprladdr	`pic S9(9)`	`input`

is a pointer to the SQL collation or collation object.

A SQL/MP Sample Database

This appendix describes the NonStop SQL/MP sample database and sample application that are included on the product site update tape (SUT). Many examples in this manual (as well as other SQL/MP manuals) refer to the sample database. You can create your own copy of the sample database and access it using SQLCI commands or by embedding SQL statement in a host-language program.

The sample database includes the PERSNL, SALES, and INVENT subvolumes. Each subvolume contains a catalog and these tables:

PERSNL	EMPLOYEE, JOB, and DEPT tables, which hold personnel data.
SALES	CUSTOMER, ORDERS, ODETAIL, and PARTS tables, which are used for order data. Also, the SUPPKANJ table, which accepts Kanji data for the supplier's name and address.
INVENT	SUPPLIER, PARTSUPP, PARTLOC, and ERRORS tables, which hold inventory data. (PARTLOC can be partitioned over three volumes, if they are available.)

The sample application demonstrates the use of SQL/MP in a Pathway environment. It includes requestors written in SCREEN COBOL and servers written in C, COBOL, Pascal, and TAL. The servers use embedded SQL statements to access the sample database.

HP distributes the sample database and application in the ZTSQLMSG subvolume. (Ask your database administrator or system manager for the volume where the ZTSQLMSG subvolume is installed on your system.)

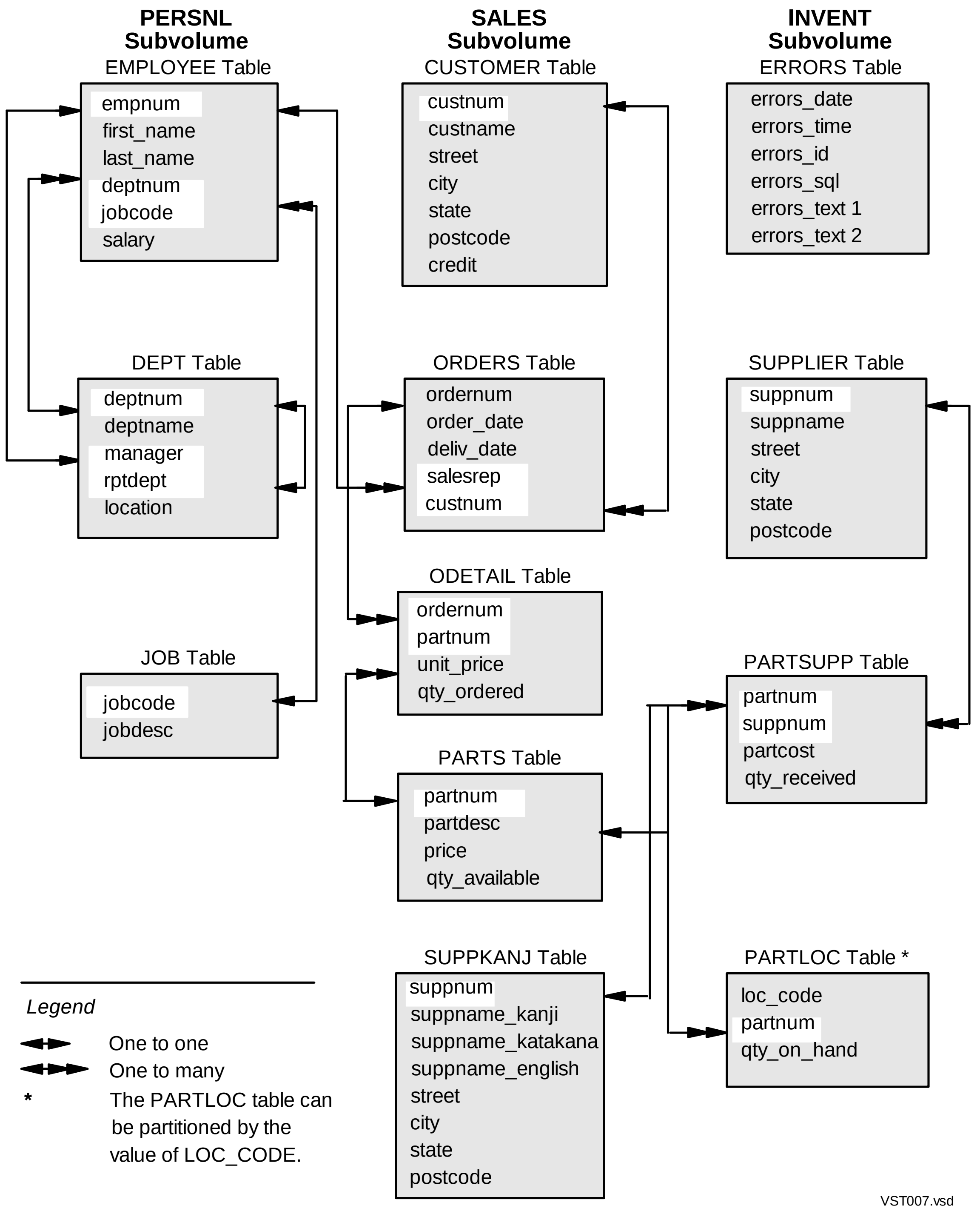
The ZTSQLMSG.DOCUMENT file describes the files in the ZTSQLMSG subvolume. The DOCUMENT file also explains how to create a copy of the sample database and how to compile and run the sample application. To print the DOCUMENT file, enter this command at the TACL prompt:

```
10> TGAL / IN ZTSQLMSG.DOCUMENT, OUT $S.#loc /
```

The *loc* parameter is a spooler location for your system.

Figure A-1 shows the names of columns and tables and the relations between the tables in the sample database.

Figure A-1. SQL/MP Sample Database Relations



Example A-1 shows the COPYLIB file containing the record descriptions of the sample database tables. This file was generated using INVOKE directives executed from SQLCI. For example, this INVOKE directive generates the DEPT table:

```
INVOKE PERSNL.DEPT FORMAT COBOL85 TO COPYLIB (DEPT) ;
```

For more information about SQLCI, see the *SQL/MP Reference Manual*.

The COPYLIB was edited to change the descriptions generated from VARCHAR columns to fixed-length data item descriptions, which are more appropriate for COBOL applications. For a description of the SUPPKANJ table, see the ZTSQLMSG.DOCUMENT file.

Example A-1. COPYLIB File for Sample Database (page 1 of 3)

```
*****************************************************
*   Personnel (PERSNL)                              *
*****************************************************
?SECTION EMPLOYEE
* Record Definition for \SYS1.$VOL1.PERSNL.EMPLOYEE
* Definition current at 17:09:06 - 10/10/94
 01 EMPLOYEE.
    02 EMPNUM                          PIC 9(4) COMP.
    02 FIRST-NAME                      PIC X(15).
    02 LAST-NAME                       PIC X(20).
    02 DEPTNUM                         PIC 9(4) COMP.
    02 JOBCODE                         PIC 9(4) COMP.
    02 SALARY                          PIC 9(6)V9(2) COMP.
?SECTION DEPT
* Record Definition for \SYS1.$VOL1.PERSNL.DEPT
* Definition current at 17:09:07 - 10/10/94
 01 DEPT.
    02 DEPTNUM                         PIC 9(4) COMP.
    02 DEPTNAME                        PIC X(12).
    02 MANAGER                         PIC 9(4) COMP.
    02 RPTDEPT                         PIC 9(4) COMP.
*   02 LOCATION.      <-- VARCHAR edited to be fixed-length
item.
*      03 LEN     PIC S9(4) COMP.
*      03 VAL     PIC X(18).
    02 LOCATION                        PIC X(18).
?SECTION JOB
* Record Definition for \SYS1.$VOL1.PERSNL.JOB
* Definition current at 17:09:09 - 10/10/94
 01 JOB.
    02 JOBCODE                         PIC 9(4) COMP.
*   02 JOBDESC.       <-- VARCHAR edited to be fixed-length
item.
*      03 LEN  PIC S9(4) COMP.
*      03 VAL  PIC X(18).
    02 JOBDESC                         PIC X(18).
```

Example A-1. COPYLIB File for Sample Database (page 2 of 3)

```
*****************************************************
*   Sales (SALES)                                   *
*****************************************************
?SECTION CUSTOMER
* Record Definition for \SYS1.$VOL1.SALES.CUSTOMER
* Definition current at 17:09:10 - 10/10/94
 01 CUSTOMER.
    02 CUSTNUM                         PIC 9(4) COMP.
    02 CUSTNAME                        PIC X(18).
    02 STREET                          PIC X(22).
    02 CITY                            PIC X(14).
    02 STATE                           PIC X(12).
    02 POSTCODE                        PIC X(10).
    02 CREDIT                          PIC X(2).
?SECTION ORDERS
* Record Definition for \SYS1.$VOL1.SALES.ORDERS
* Definition current at 17:09:11 - 10/10/94
 01 ORDERS.
    02 ORDERNUM                        PIC 9(6) COMP.
    02 ORDER-DATE                      PIC S9(6) COMP.
    02 DELIV-DATE                      PIC S9(6) COMP.
    02 SALESREP                        PIC 9(4) COMP.
    02 CUSTNUM                         PIC 9(4) COMP.
?SECTION ODETAIL
* Record Definition for \SYS1.$VOL1.SALES.ODETAIL
* Definition current at 17:09:12 - 10/10/94
 01 ODETAIL.
    02 ORDERNUM                        PIC 9(6) COMP.
    02 PARTNUM                         PIC 9(4) COMP.
    02 UNIT-PRICE                      PIC S9(8)V9(2) COMP.
    02 QTY-ORDERED                     PIC 9(5) COMP.
?SECTION PARTS
* Record Definition for \SYS1.$VOL1.SALES.PARTS
* Definition current at 17:09:13 - 10/10/94
 01 PARTS.
    02 PARTNUM                         PIC 9(4) COMP.
    02 PARTDESC                        PIC X(18).
    02 PRICE                           PIC S9(8)V9(2) COMP.
    02 QTY-AVAILABLE                   PIC S9(7) COMP.
```

Example A-1. COPYLIB File for Sample Database (page 3 of 3)

```
*****************************************************
*   Inventory (INVENT)                              *
*****************************************************
?SECTION SUPPLIER
* Record Definition for \SYS1.$VOL1.INVENT.SUPPLIER
* Definition current at 17:09:14 - 10/10/94
 01 SUPPLIER.
    02 SUPPNUM                           PIC 9(4) COMP.
    02 SUPPNAME                          PIC X(18).
    02 STREET                            PIC X(22).
    02 CITY                              PIC X(14).
    02 STATE                             PIC X(12).
?SECTION PARTSUPP
* Record Definition for \SYS1.$VOL1.INVENT.PARTSUPP
* Definition current at 17:09:15 - 10/10/94
 01 PARTSUPP.
    02 PARTNUM                           PIC 9(4) COMP.
    02 SUPPNUM                           PIC 9(4) COMP.
    02 PARTCOST                          PIC S9(8)V9(2) COMP.
    02 QTY-RECEIVED                      PIC 9(5) COMP.
?SECTION PARTLOC
* Record Definition for \SYS1.$VOL1.INVENT.PARTLOC
* Definition current at 17:09:16 - 10/10/94
 01 PARTLOC.
    02 LOC-CODE                          PIC X(3).
    02 PARTNUM                           PIC 9(4) COMP.
    02 QTY-ON-HAND                       PIC S9(7) COMP.
?SECTION ERRORS
*  Record Definition for table \SYS1.$VOL1.INVENT.ERRORS
*  Definition current at 17:09:16 - 10/10/94
 01 ERRORS.
    02 ERRORS-DATE                       PIC 9(6) COMP.
    02 ERRORS-TIME                       PIC 9(6) COMP.
    02 ERRORS-ID                         PIC 9(6) COMP.
    02 ERRORS-SQL                        PIC 9(4) COMP.
    02 ERRORS-TEXT1                      PIC X(240).
    02 ERRORS-TEXT2                      PIC X(240).
```

B Memory Considerations

This appendix describes the SQL internal data structures generated in a COBOL program, including this information:

- Using the SQLMEM directive to control the placement of the SQL internal data structures
- Estimating the memory required by a COBOL program

Topics include:

SQL/MP Internal Structures

The HP COBOL compiler generates the SQLINALL internal data structure to maintain information about the SQL statements and host variables used in the program. Depending on the statements used in a program, SQLINALL can contain these substructures:

- SQLIN*n* for each SQL statement that generates a call to the SQL executor
- SQLVARS*n*I for each input host variable in the program
- SQLVARS*n*O for each output host variable in the program

Each SQL structure or substructure name includes an identification number (*n*) assigned by the HP COBOL compiler. Although you cannot directly manipulate these structures, you can control their placement in memory and avoid using their names in your own structures.

The system automatically creates an extended data segment for each COBOL program at run time and places the SQL internal structures in this extended data segment, even if you do not declare an Extended-Storage Section in your program. If your program requires the SQL internal structures to be placed in the user data segment (for example, for a program that runs as a process pair), see Using the SQLMEM Directive on page B-4.

All Extended-Storage Sections in a main program and in any subprograms called by the main program are automatically consolidated into the single extended data segment allocated by the main program. You do not need to use the Binder program to perform this function, and you do not need to allocate space in your data stack or your Extended-Storage Section for these data structures.

Resizing Segments

The SQL executor and any utilities that use the SQLINALL data structure use Guardian procedures for resizing an existing extended segment. Therefore, you do not need to be concerned about the size of the extended data segments that NonStop SQL/MP uses to store the SQLINALL data structure.

If an executing program requires more memory, SQL/MP automatically increases the size of extended data segments. As a result, different listings from different executions of the same program could show different memory sizes. This difference reflects the dynamic memory management provided by SQL/MP.

The HP COBOL compiler initializes all the pointers in the SQLINALL structure in the SQL-INIT section, which the compiler writes at the end of the source program. To initialize the pointers, the compilers insert a call to the SQLADDR procedure.

Avoiding Name Conflicts

In a COBOL program, avoid using names that conflict with the names in the internal SQL data structures. The section names generated by the HP COBOL compiler in your program are SQLDO-*n* (where *n* is an integer) and SQL-INIT. These examples show the names used in internal SQL structures.

SQLINALL Structure

The SQLINALL structure is shown in the next example. The HP COBOL compiler generates an SQLIN*n* substructure for each SQL statement that causes a call to the SQL executor.

```
01 SQLINALL-T9192-TANDEM.
   05 STMT-PROCEDURE-ID         PIC X(32) VALUE "proc-name".
   05 SQLINIT-FLAG              PIC S9(4) COMP VALUE -1.
   05 SQLINn
      10 EYE-CATCHER            PIC X(2)  VALUE "IN".
      10 VERSION                PIC S9(4) COMP VALUE 0.
      10 SLT-INDEX              PIC S9(4) COMP VALUE index.
      10 PROCEDURE-ID           PIC X(32) VALUE "proc-name".
      10 USER-LINE-NUMBER       PIC S9(9) COMP VALUE line.
      10 STMT-PROCEDURE-ID-PTR  PIC S9(9) COMP VALUE -999999.
      10 SRCFILE-PTR            PIC S9(9) COMP VALUE -999999.
      10 OPCODE                 PIC S9(4) COMP VALUE opcode.
      10 FLAGS                  PIC S9(4) COMP VALUE 0.
      10 P-CKSUM                PIC S9(4) COMP VALUE 0.
      10 IOVAR-CKSUM            PIC S9(9) COMP VALUE 0.
      10 IVARS-PTR              PIC S9(9) COMP VALUE -999999.
      10 OVARS-PTR              PIC S9(9) COMP VALUE -999999.
      10 SQLSA-PTR              PIC S9(9) COMP VALUE -999999.
```

If the compiler cannot determine the procedure name, PROCEDURE-ID is replaced by this declaration, and the executor provides the name later.

```
      10 BVARS-PTR           PIC S9(9) COMP VALUE -999999.
      10 FILLER              PIC X(27).
```

SQLVARS Structure

If the SQL statement contains input or output host variables, the HP COBOL compiler creates an SQLVARS*n*I structure for each input variable and an SQLVARS*n*O structure for each output variable:

```
05 SQLVARSn{I|O}.
   10 EYE-CATCHER          PIC X(2)   VALUE "D1".
   10 NUM-ENTRIES          PIC S9(4) COMP VALUE num-entries.
   10 VARSn.
      15 DATA-TYPE         PIC S9(4) COMP VALUE datatype.
      15 DATA-LEN          NATIVE-2 VALUE length.
      15 PRECISION         PIC S9(4) COMP VALUE precision.
      15 NULL-INFO         PIC S9(4) COMP VALUE null-flag.
      15 VAR-PTR           PIC S9(9) COMP VALUE -999999.
      15 IND-PTR           PIC S9(9) COMP VALUE -999999.
      15 CPRL-PTR          PIC S9(9) COMP VALUE -999999.
      15 RESERVED          PIC S9(9) COMP VALUE -1.
```

For a host variable with a numeric, date-time, or interval data type, DATA-LEN is replaced by this declaration:

```
      15 DATA-SCALEN       PIC S9(4) COMP VALUE length.
```

Depending on the data type of the host variable, the DATA-SCALEN data item has these values:

Data Type of Host Variable	**Value of DATA-SCALEN**
Numeric	Byte 1: scale Byte 2: length
Date-time or interval	Byte 1: Date-time qualifier Byte 2: length

For a host variable with a character, date-time or interval data type, PRECISION has these values:

Data Type of Host Variable	**Value of PRECISION**
Numeric	Zero (0)
Date-time or interval	Byte 1: leading field precision Byte 2: fractional precision
Character	Character set ID

SQLBVARS Structure

If the SLT-INDEX item of SQLIN*n* is -1, the HP COBOL compilers creates an SQLBVARS entry. This entry occurs when a qualified cursor or statement name has a qualifying program name other than the current program name. The SQLBVARS entry follows the declaration of its corresponding SQLIN:

```
05 SQLBVARSn.
   10 EYE-CATCHER            PIC X(2)  VALUE "VB".
   10 NUM-ENTRIES            PIC S9(4) COMP VALUE 2.
   10 BVARS1.
      15 DATA-TYPE           PIC S9(4) COMP VALUE proc-type.
      15 DATA-LEN            PIC S9(4) COMP VALUE length.
      15 RESERVED0           PIC S9(4) COMP VALUE 0.
      15 VAR-PTR             PIC S9(9) COMP VALUE -999999.
      15 IND-PTR             PIC S9(9) COMP VALUE -999999.
   10 BVARS2.
      15 DATA-TYPE           PIC S9(4) COMP VALUE cursor-type.
      15 DATA-LEN            PIC S9(4) COMP VALUE length.
      15 RESERVED0           PIC S9(4) COMP VALUE 0.
      15 VAR-PTR             PIC S9(9) COMP VALUE -999999.
      15 IND-PTR             PIC S9(9) COMP VALUE -999999.
05 SQLBVARS-PROC-NAMEn      PIC X(32) VALUE procedure-name.
05 SQLBVARS-CUR-STMT-NAMEn PIC X(30) VALUE statement-name.
```

Using the SQLMEM Directive

The SQLMEM compiler directive specifies where in memory (user data segment or extended data segment) the HP COBOL compiler should place the internal SQL data structures. Although a program does not explicitly declare the SQL internal data structures and cannot directly access them, it can control their placement in memory.

Specify the SQLMEM directive before the Data Division in a COBOL program using this syntax:

```
SQLMEM { USER | EXT }
```

`USER`

directs the compiler to place the SQL data structures in the user data segment (Working-Storage Section) even if the program includes an Extended-Storage Section.

`EXT`

At run time, the system automatically creates an extended data segment for a COBOL program and places the SQL structures in this segment, even if the program does not include an Extended-Storage Section.

EXT is the default.

Note. The HP COBOL compiler does not perform checkpointing on data in an extended data segment. Therefore, to ensure data integrity, HP recommends that you use the NonStop Transaction Management Facility (TMF).

Estimating Memory Requirements

A program that uses embedded SQL statements and directives to access an SQL/MP database uses more memory than a program that accesses an Enscribe database. This subsection describes how to estimate the virtual memory used by embedded SQL statements and directives. Some statements require no extra extended memory. However, other statements generate a run-time call to the SQL executor and do use extra memory.

Memory Requirements

These structures are shared by all SQL statements and directives:

Structure	Bytes	Description
SQLCA	430	Count once if you specify the INCLUDE SQLCA directive.
SQLSA	838	Count once if you specify the INCLUDE SQLSA directive.

Use this table to estimate the number of bytes used by each embedded SQL statement and directive.

Bytes Required	Description
32	Base value for header information.
+ 72	Base value for each SQL statement with no host variables.
+ 4 + (24 x number of input host variables)	Required for a statement with input host variables.
+ 4 + (24 x number of output host variables)	Required for a statement with output host variables.
+ 146	Required for a static SQL statement that uses a cursor declared in the global area of the program.

Follow these guidelines when you estimate bytes required:

- Count a host variable once per occurrence.
- Count only these SQL statements and directives (which generate a run-time call to the SQL executor):

ALTER	DROP	INSERT
BEGIN WORK	END WORK	LOCK TABLE
CLOSE	EXECUTE	OPEN
COMMENT	EXECUTE IMMEDIATE	RELEASE
CREATE	FETCH	ROLLBACK WORK
DELETE	FREE RESOURCES	SELECT
DESCRIBE	GET VERSION	UNLOCK TABLE
DESCRIBE INPUT	HELP TEXT	UPDATE
		UPDATE STATISTICS

Do not count these SQL statements and directives:

- BEGIN DECLARE SECTION and END DECLARE SECTION
- CONTROL EXECUTOR, CONTROL QUERY, and CONTROL TABLE
- DECLARE CURSOR
- INVOKE
- WHENEVER

Guidelines for Memory Use

The system allocates real memory in 16 KB pages. If an SQL statement uses only part of a page, the system allocates the entire page. Therefore, the real memory used by embedded SQL statements can be larger than the figures shown under Memory Requirements on page B-5.

A program can encounter memory problems in these situations:

- The program contains a large number of embedded SQL statements.
- The program runs on a system with limited memory (for example, 16 MB or less).
- The program runs in a CPU that is also running a large number of other programs.

To reduce the memory use in an extended data segment, follow these guidelines:

- Declare only the host variables that your program actually requires.
- Declare all host variables in one Declare Section. The system then allocates the host variables contiguously in one or more pages, rather than allocating each host variable in a separate page.
- Execute SQL statements in listing order as often as possible. Therefore, the SQL statements can share many of the pages in the extended data segment.
- As a last measure, use dynamic SQL statements. Using dynamic SQL statements can reduce memory use. However, it can also degrade a program’s performance because of the additional SQL run-time compilations.

C Maximizing Local Autonomy

This appendix describes about the local autonomy in the NonStop SQL/MP network-distributed database.

Topics include:

- Using a Local Partition
- Using TACL DEFINEs on page C-2
- Using Current Statistics on page C-2

Local autonomy in the NonStop SQL/MP network-distributed database ensures that a program can access data on the local node, regardless of the availability of SQL objects on remote nodes. In some cases, the design of NonStop SQL/MP allows for local autonomy. For example, if a DDL change alters a table on \NODEA when \NODEB is unavailable, an SQL program file on \NODEB that uses the altered \NODEA table is not marked as invalid. The invalid SQL program on \NODEB that is erroneously marked as valid is detected at run time by the timestamp check and then automatically recompiled.

If your program accesses a network-distributed database, you can maximize local autonomy by following these guidelines:

- Use a local partition, rather than the primary partition, as the table name for partitioned tables.
- Use TACL DEFINEs to refer to tables.
- Use current statistics.
- Skip unavailable partitions.

For collations, SQL/MP supports run-time node autonomy because collations are stored in an SQL object's file label and within expressions that operate on the SQL objects. For example, suppose that you create a partitioned table, TABLEA, with partitions on \NEWYORK and \PARIS. TABLEA requires the collation \NEWYORK.$SQL.COLLATE.FRENCH. If \NEWYORK goes down, programs on \PARIS that refer to TABLEA continue to run, because they get the collation information from the TABLEA file label. However, the recompilation of a program on \PARIS that uses TABLEA fails, because the \NEWYORK.$SQL.COLLATE.FRENCH collation is not available.

Using a Local Partition

If your program uses a remote partition, the SQL compiler looks for information about the table in a remote catalog. If the remote node is down, the SQL compilation fails. However, if your program uses a local partition, the SQL compiler looks for the information in a local catalog. If the local node and the data is available, the SQL compilation is successful.

This example shows the concept of maximizing local autonomy. The PARTS table is a partitioned table that resides on the \NEWYORK and \PARIS nodes:

\NEWYORK	The first partition contains all rows in which PARTS.PARTNUM (the primary key) is less than 5000.
\PARIS	The second partition contains all rows in which PARTS.PARTNUM is 5000 or greater. An index on the PARTDESC column of table PARTS, is named IXPART.

A program declares an SQL cursor as follows:

```
EXEC SQL DECLARE GET_PART_CURSOR CURSOR FOR
  SELECT PARTNUM, PARTDESC, PRICE, QTY_AVAILABLE
    FROM  =PARTS
    WHERE PARTS.PARTNUM < 5000
      AND PARTS.PARTDESC = "V8 DISK OPTION"
END-EXEC.
```

The program running on \NEWYORK uses a DEFINE to associate the PARTS table with the first partition located at \NEWYORK**:**

```
SET DEFINE CLASS MAP
ADD DEFINE =parts,    FILE \NEWYORK.$VOL1.SALES.PARTS
```

If \PARIS is unavailable at compile time, the program can still compile because enough information is available in the catalogs on \NEWYORK, where the first partition is registered.

Suppose that the compiler uses the index on \PARIS in the optimized execution plan. If \PARIS is still unavailable at run time, the executor invokes the compiler to automatically recompile the statement. The compiler determines an execution plan that does not use the index IXPART but will sequentially scan the rows in the first partition to find all parts that have “V8 DISK OPTION” in the PARTDESC column.

Using TACL DEFINEs

Use class MAP DEFINEs in a program to refer to tables. By associating DEFINEs with local partitions rather than remote partitions, you can increase the number of successful compilations of the programs that access a distributed database. All SQL compilations are affected, including both explicit compilations and automatic recompilations.

Using Current Statistics

For a partitioned table to have local autonomy, the UPDATE STATISTICS statement must be executed on the table at least once. If the SQL catalog in which a table is registered does not have any statistics for the table, the SQL optimizer does a catalog look-up operation for each partition of the table to estimate the aggregate number of nonempty blocks and records. Also, if the statistics for an unavailable partitioned table have not been updated, you will receive an SQL warning and file-system error even if your query does not try to retrieve any rows from the unavailable partition. Executing the UPDATE STATISTICS statement can eliminate both these problems.

Skipping Unavailable Partitions

Use the SKIP UNAVAILABLE PARTITION option of the CONTROL TABLE directive to cause SQL/MP to skip a partition that is not available and to open the next available partition that satisfies the search condition of a query. (SQL/MP also returns warning message 8239 to the SQLCA structure.) The SKIP UNAVAILABLE PARTITION option applies to static or dynamic SQL statements that refer to partitioned tables and partitioned indexes of the tables.

D Converting COBOL Programs

This appendix describes how a COBOL program developed for NonStop SQL/MP version 1 or version 2 software can execute on SQL/MP version 300 (or later) software without changes to its embedded SQL statements or directives. However, to use new SQL features, you must modify and recompile the program.

Topics include:

- Generating SQL Data Structures
- Generating SQLDA Structures on page D-2
- Planning for Future PVUs on page D-8

Generating SQL Data Structures

The SQLCA, SQLSA, and SQLDA data structures can change in new PVUs of SQL/MP. Table D-1 lists the SQL data structure and the changes that occurred with each version.

Table D-1. SQL Data Structures

Version	Size, Bytes	Eye-Catcher	Literals	New Fields
SQLCA Structure				
1	430	CA	None	–
2	430	CA	None	None
Š 300	430	CA	None	None
SQLSA Structure				
1	838	SA	None	–
2	838	SA	None	None
Š 300	838	SA	None	OUTPUT–COLLATIONS–LEN
SQLDA Structure				
1	Variable	DA	None	–
2	Variable	D1	None	PRECISION, NULL–INFO, and IND–PTR
Š 300	Variable	D1	None	CPRL–PTR, user-defined collation buffer

Follow these guidelines if you are converting an existing COBOL program (that is, a program that uses version 1 or version 2 structures) or writing a new program to use version 300 (or later) SQL structures:

- Use the INCLUDE STRUCTURES directive to specify the version of each structure you require, even if you require version 1 or version 2 structures. To generate version 300 or later SQL data structures, you must use the

INCLUDE STRUCTURES directive. For more information, see Section 9, Error and Status Reporting.

- Use the SQLCAGETINFOLIST procedure to return information from the SQLCA structure. Do not access this structure directly. HP reserves the right to change it in future PVUs.

Generating SQLDA Structures

If your existing COBOL program generates SQLDA structures and you are converting it to run on version 300 (or later) SQL/MP software, you might need one or more of these combinations of SQLDA structures:

- A version 300 (or later) SQLDA structure
- A version 1 or version 2 SQLDA structure
- A version 300 (or later) SQLDA structure and a version 1 or version 2 SQLDA structure

Generating a Version 315 SQLDA Structure

To convert a program that generates a version 1 or version 2 SQLDA structure to generate a version 300 (or later) SQLDA structure, follow these steps:

1. If necessary, remove the RELEASE1 or RELEASE2 option from the SQL compiler directive or from the INCLUDE SQLDA directive. The HP COBOL compiler returns an error if you specify the RELEASE1 or RELEASE2 option and the INCLUDE STRUCTURES directive.
2. Remove any -R1 or -R2 suffixes appended to SQLDA field names.
3. If you are converting a version 1 SQLDA structure, make sure you initialize the NULL-INFO and IND-PTR fields.
4. Add an INCLUDE STRUCTURES directive with the ALL VERSION 315 option:

   ```
   EXEC SQL INCLUDE STRUCTURES ALL VERSION 315 END-EXEC.
   ```

   Or specify only the SQLDA VERSION 315 option:

   ```
   EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 315 END-EXEC.
   ```
5. Add the necessary Procedure Division statements to process the version 315 SQLDA structure. For the layout of a version 315 SQLDA structure and a description of each field, see Section 10, Dynamic SQL Operations.

Generating a Version 2 SQLDA Structure

If you are converting a program to use the INCLUDE STRUCTURES directive, but you require a version 2 SQLDA structure, follow these steps:

1. If necessary, remove the RELEASE2 option from the SQL compiler directive or the INCLUDE SQLDA directive. The HP COBOL compiler returns an error if you specify the RELEASE2 option and the INCLUDE STRUCTURES directive.
2. If you specified the RELEASE2 option in an INCLUDE SQLDA directive, remove any -R2 suffixes you appended to SQLDA field names.
3. If you are converting a version 1 SQLDA structure, initialize the NULL-INFO and IND-PTR fields. (Your program should already initialize these fields for a version 2 SQLDA structure.)
4. Add an INCLUDE STRUCTURES directive with the ALL VERSION 2 option:

   ```
   EXEC SQL INCLUDE STRUCTURES ALL VERSION 2 END-EXEC.
   ```

   Or specify only the SQLDA VERSION 2 option:

   ```
   EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 2 END-EXEC.
   ```

Example D-1 shows a version 2 SQLDA structure.

Example D-1. Version 2 SQLDA Structure

```
01 sqlda-name.
   05 EYE-CATCHER       PIC X(2) VALUE "D1".
   05 NUM-ENTRIES       PIC S9(4) COMP VALUE sqlvar-count.
   05 SQLVAR [ -R2 ]    OCCURS sqlvar-count TIMES.
      10 DATA-TYPE      PIC S9(4) COMP.
      10 DATA-LEN       NATIVE-2.
      10 PRECISION      PIC S9(4)  COMP.
      10 NULL-INFO      PIC S9(4)  COMP..
      10 VAR-PTR        PIC S9(9)  COMP.
      10 IND-PTR        PIC S9(9)  COMP.
      10 RESERVED       PIC S9(18) COMP.

01  names-buffer-name PIC X( length ).
```

Table D-2 describes each field in a version 2 SQLDA structure.

Table D-2. Version 2 SQLDA Structure Fields (page 1 of 3)

Field Name	Description
EYE-CATCHER	An identifying field your program must initialize as D1 for version 1 or DA for version 2. SQL/MP statements do not return values to EYE-CATCHER.
NUM-ENTRIES	The number of input or output parameters the SQLDA structure can accommodate.
SQLVAR	Group item that describes input parameters or database columns. The DESCRIBE INPUT and DESCRIBE statements return one SQLVAR entry for each input parameter or each output variable.
DATA-TYPE	The data type of the parameter or output variable.

Table D-2. Version 2 SQLDA Structure Fields (page 2 of 3)

Field Name	**Description**	
DATA-LEN	DATA-LEN depends on the data type.	
	Fixed-length character	The number of bytes in the string.
	Variable-length character	The maximum number of bytes in the string.
	Decimal numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item.
	Binary numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item (2, 4, or 8).
	Date-time or INTERVAL	Bits 0:7 contain one of these codes for the range of date-time fields. Bits 8:15 contain the storage size of the item.

1	Year to Year	11	Year to Minute	20	Day to Minute
2	Month to Month	12	Year to Second	21	Day to Second
3	Day to Day	13	Year to Fraction	22	Day to Fraction
4	Hour to Hour	14	Month to Day	23	Hour to Minute
5	Minute to Minute	15	Month to Hour	24	Hour to Second
6	Second to Second	16	Month to Minute	25	Hour to Fraction
7	Fraction to Fraction	17	Month to Second	26	Minute to Second
8	Year to Month	18	Month to Fraction	27	Minute to Fraction
9	Year to Day	19	Day to Hour	28	Second to Fraction
10	Year to Hour				

PRECISION	The PRECISION value depends on the data type.	
	Binary numeric	PRECISION contains the numeric precision.
	Date-time or INTERVAL	Bits 0:7 contain the leading field precision. Bits 8:15 contain the fraction precision. If the FRACTION field is not included, bits 8:15 are 0.

Table D-2. Version 2 SQLDA Structure Fields (page 3 of 3)

Field Name	Description
NULL-INFO	For input parameters, NULL-INFO contains a negative integer if the column permits null values. For output columns, NULL-INFO contains a negative integer if the row returned is null.
VAR-PTR	The extended address of the actual data (value of input parameter or column). SQL/MP does not return VAL-PTR. Your program must initialize VAR-PTR to point to the input and output data buffers.
IND-PTR	The address of a flag that indicates whether a parameter or column is actually null. ● For input parameters, your program initializes IND-PTR to -1 if the user entered a null value. ● For output columns, SQL initializes the location IND-PTR points to -1 if the user entered a null value. If your program does not need to process null values, initialize IND-PTR to an invalid address.

Generating a Version 1 SQLDA Structure

If you are converting a program to use the INCLUDE STRUCTURES directive, but you require a version 1 SQLDA structure, follow these steps:

1. If necessary, remove the RELEASE1 option from the SQL compiler directive or the INCLUDE SQLDA directive. The HP COBOL compiler returns an error if you specify the RELEASE1 option and the INCLUDE STRUCTURES directive.
2. If you specified the RELEASE1 option in an INCLUDE SQLDA directive, remove any -R1 suffixes you appended to SQLDA field names.
3. Add an INCLUDE STRUCTURES directive with the ALL VERSION 1 option:

   ```
   EXEC SQL INCLUDE STRUCTURES ALL VERSION 1 END-EXEC.
   ```

   Or specify only the SQLDA VERSION 1 option:

   ```
   EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 1 END-EXEC.
   ```

Example D-2 shows a version 1 SQLDA structure.

Example D-2. Version 1 SQLDA Structure

```
01 sqlda-name.
   05 EYE-CATCHER       PIC X(2) VALUE "DA".
   05 NUM-ENTRIES       PIC S9(4)  COMP.
   05 SQLVAR [ -R1 ]    OCCURS  sqlvar-count TIMES.
      10  DATA-TYPE     PIC S9(4)  COMP.
      10  DATA-LEN      NATIVE-2.
      10  RESERVED-0    PIC S9(4)  COMP.
      10  VAR-PTR       PIC S9(9)  COMP.
      10  RESERVED      PIC S9(9)  COMP.
01  names-buffer-name  PIC X( length ).
```

Table D-3 describes each field in a version 1 SQLDA structure.

Table D-3. Version 1 SQLDA Structure Fields (page 1 of 2)

Field Name	Description
EYE-CATCHER	An identifying field that a program must initialize as D1 for version 1 or DA for version 2. SQL/MP statements do not return values to EYE-CATCHER.
NUM-ENTRIES	The number of input parameters or output variables the SQLDA structure can accommodate.
SQLVAR	Group item that describes input parameters or database columns. The DESCRIBE INPUT and DESCRIBE statements return one SQLVAR entry for each input parameter or each output variable.

Table D-3. Version 1 SQLDA Structure Fields (page 2 of 2)

Field Name	**Description**		
DATA-TYPE	The data type of the parameter or output variable.		
DATA-LEN	The DATA-LEN value depends on the data type.		
	Fixed-length character	The number of bytes in the string.	
	Variable-length character	The maximum number of bytes in the string.	
	Decimal numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item.	
	Binary numeric	Bits 0:7 contain the decimal scale. Bits 8:15 contain the byte length of the item (2, 4, or 8).	
	Date-time or INTERVAL	Bits 0:7 contain one of these codes for the range of the field. Bits 8:15 contain the storage size of the item.	
	1 Year to Year	11 Year to Minute	20 Day to Minute
	2 Month to Month	12 Year to Second	21 Day to Second
	3 Day to Day	13 Year to Fraction	22 Day to Fraction
	4 Hour to Hour	14 Month to Day	23 Hour to Minute
	5 Minute to Minute	15 Month to Hour	24 Hour to Second
	6 Second to Second	16 Month to Minute	25 Hour to Fraction
	7 Fraction to Fraction	17 Month to Second	26 Minute to Second
	8 Year to Month	18 Month to Fraction	27 Minute to Fraction
	9 Year to Day	19 Day to Hour	28 Second to Fraction
	10 Year to Hour		
VAR-PTR	The extended address of the actual data (value of input parameter or column). SQL/MP does not return VAL-PTR. Your program must initialize it to point to the input and output data buffers.		

Using a Combination of SQLDA Structures

Version 300 (or later) SQL/MP does not support different versions of SQLDA structures in the same compilation unit. If your program requires more than one SQLDA structure in a compilation unit, convert all SQLDA structures to version 315. However, if you want to use a combination of SQLDA structures (for example, a version 2 structure and a version 315 structure), follow these steps:

1. Separate your program into different compilation units so that the version 315 SQLDA structure and the supporting Procedure Division statements are in a different compilation unit than the version 2 (or version 1) SQLDA structure and Procedure Division statements.
2. Specify an INCLUDE STRUCTURES directive with the appropriate VERSION clause in each compilation unit.
3. Compile each compilation unit separately.
4. Use the Binder program to combine the object files into a single target object file.

Planning for Future PVUs

If you are converting a COBOL program developed for SQL/MP version 1 or version 2 software to use version 300 (or later) features and to run on SQL/MP version 300 (or later) software, consider making these changes in your program for compatibility with future SQL/MP PVUs.

SQL/MP Version Procedures

The SQLGETOBJECTVERSION, SQLGETCATALOGVERSION, and SQLGETSYSTEMVERSION system procedures, which return SQL version information, might not be supported in a future PVU and will return an “Unresolved External References” error at run time.

If you call any of these procedures, consider modifying your program.

Procedure	**Description of Conversion**
SQLGETOBJECTVERSION	Convert to the GET VERSION statement, or query the TABLES.OBJECTVERSION column.
SQLGETCATALOGVERSION	Convert to the GET VERSION OF CATALOG statement, or query the VERSIONS.CATALOGVERSION column.
SQLGETSYSTEMVERSION	Convert to the GET VERSION OF SYSTEM statement.

For more information, including the syntax of the GET VERSION statements, see the *SQL/MP Reference Manual*.

RELEASE1 and RELEASE2 Options

The RELEASE1 and RELEASE2 options used in the SQL compiler directive and the INCLUDE SQLDA directive might not be supported in a future PVU.

Consider modifying your program to use the INCLUDE STRUCTURES directive with the VERSION 1 or VERSION 2 options to generate version 1 or version 2 SQLDA structures. Or, convert your program to use version 300 (or later) SQLDA structures. Remove the RELEASE1 or RELEASE2 option from the SQL compiler directive or the INCLUDE SQLDA directive.

For more information about the INCLUDE STRUCTURES directive, see Using the INCLUDE STRUCTURES Directive on page 9-1.

E Writing Pathway Servers

Writing a Pathway application that accesses a NonStop SQL/MP database is similar to writing an application that accesses an Enscribe database. In either case, you manage terminal requests and displays in the requester portion of the application and perform database manipulation in the servers. When coding a Pathway application that accesses an SQL/MP database, you embed the SQL DML statements in the server code.

Most Pathway servers use subroutines. Subroutines are efficient because they reduce the amount of coding to a minimum. Also, subroutines separate the database manipulation operations from the main logic, which makes it easier to develop and maintain programs.

This appendix provides sample program code to illustrate two possible models for writing Pathway servers that use subroutines to separate the database manipulation operations from the main logic.

- PERFORM model. This model uses COBOL PERFORM statements in the main logic flow of a program to execute subroutines in the same program. When the PERFORM is executed, the program enters the subroutine, performs the requested function, and returns control to the next instruction in the main logic.
- CALL model. This model uses COBOL CALL statements in the main program to call subprograms. In this model, a program consists of a collection of subprograms managed by a main program.

  If the main program passes arguments to the subprogram, the subprogram must contain a Linkage Section. If no arguments are passed, the subprogram need not contain a Linkage Section. In HP COBOL, a main program cannot contain a Linkage Section.

PERFORM Model

This PERFORM model Pathway server program modifies a single file, called PARTS in the sample database, depending on the value of the ENTRY-TYPE flag of the message received. When the flag is U, the server updates the inventory count for the specified part number. When the flag is I, it inserts a new PARTS record.

The reply message fields REPLY-CODE and ERROR-CODE are set as follows:

Reply Message	**REPLY-CODE**	**ERROR-CODE**
Operation successful	0	N.A.
Error, operation not performed	9999	Positive SQLCODE value (transformed from original negative value)
Warning, but operation successful	9998	Positive SQLCODE value
Invalid ENTRY-TYPE	9997	N.A.

The sample code for the PERFORM model server is shown in Example E-1.

Example E-1. PERFORM Model (page 1 of 3)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.  perform-model-sql.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER.  Tandem NonStop.
   OBJECT-COMPUTER.  Tandem NonStop.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT msg-in
         ASSIGN TO $RECEIVE
         FILE STATUS IS receive-file-status.
       SELECT msg-out
         ASSIGN TO $RECEIVE
         FILE STATUS IS receive-file-status.

 DATA DIVISION.
 FILE SECTION.
 FD  msg-in
     LABEL RECORDS ARE OMITTED.
* The definition of ENTRY-MSG should be a COPY statement.
      01 entry-msg.
         02 pw-header.
            04 reply-code         PIC S9(4) COMP.
            04 application-code   PIC XX.
            04 function-code      PIC XX.
            04 trans-code         PIC 99.
            04 term-id            PIC X(15).
            04 log-request        PIC X.
         02 entry-type            PIC X.
         02 parts-info.
            04 partnum            PIC 9(4).
            04 partname           PIC X(18).
            04 inventory          PIC S999 COMP.
            04 location           PIC XXX.
            04 price              PIC 9(6)V99 COMP.

 FD  msg-out
     LABEL RECORDS ARE OMITTED.
     RECORD CONTAINS 1 TO 26 CHARACTERS.
* The definition of ENTRY-REPLY should be a COPY statement.
   01 entry-reply.
      02 pw-header.
         04 reply-code         PIC S9(4) COMP.
         04 filler             PIC X(22).
      02 error-code            PIC S9(4) COMP.
```

Example E-1. PERFORM Model (page 2 of 3)

```
 WORKING-STORAGE SECTION.
   01 receive-file-status.
      02 stat-1                PIC 9.
         88 close-from-requester  VALUE 1.
      02 stat-2                PIC 9.

     EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.

* The definition of PARTS-RECORD should be an INVOKE directive.

   01 parts-record.
      02 partnum               PIC 9(4).
      02 partname              PIC X(18).
      02 inventory             PIC S999 COMP.
      02 location              PIC XXX.
      02 price                 PIC 9(6)V99 COMP.

     EXEC SQL  END DECLARE SECTION  END-EXEC.

     EXEC SQL  INCLUDE SQLCA  END-EXEC.

 PROCEDURE DIVISION.
 MAIN-SECTION SECTION.
 00-whenever.
     EXEC SQL  WHENEVER NOT FOUND  PERFORM :sql-notfnd   END-
EXEC.
     EXEC SQL  WHENEVER SQLERROR   PERFORM :sql-error    END-
EXEC.
     EXEC SQL  WHENEVER SQLWARNING PERFORM :sql-warning  END-
EXEC.

 a-init.
     OPEN INPUT msg-in.
     OPEN OUTPUT msg-out SYNCDEPTH 1.
     PERFORM b-trans UNTIL close-from-requester.
    STOP RUN.

 b-trans.
     MOVE SPACES to entry-reply, entry-msg.
     MOVE ZERO to reply-code OF entry-reply.
     READ msg-in
          AT END STOP RUN
     END-READ.
     MOVE pw-header OF msg-in TO pw-header OF msg-out.
     IF entry-type = "U" THEN
       PERFORM update-parts
     ELSE IF entry-type = "I" THEN
       PERFORM insert-parts
     ELSE
       MOVE 9997 TO reply-code OF entry-reply
     END-IF.
     WRITE entry-reply END-WRITE.
```

Example E-1. PERFORM Model (page 3 of 3)

```
 update-parts.
     MOVE partnum   OF parts-info TO partnum   OF parts-record.
     MOVE inventory OF parts-info TO inventory OF parts-record.

     EXEC SQL  UPDATE $mkt.sample.parts
               SET inventory = :parts-record.inventory
               WHERE partnum = :parts-record.partnum
     END-EXEC.

 insert-parts.
     MOVE partnum   OF parts-info TO partnum   OF parts-record.
     MOVE partname  OF parts-info TO partname  OF parts-record.
     MOVE inventory OF parts-info TO inventory OF parts-record.
     MOVE location  OF parts-info TO location  OF parts-record.
     MOVE price     OF parts-info TO price     OF parts-record.

     EXEC SQL  INSERT INTO $mkt.sample.parts
               VALUES ( :parts-record.partnum,
                        :parts-record.partname,
                        :parts-record.inventory,
                        :parts-record.location,
                        :parts-record.price )
     END-EXEC.

 sql-error.
* Return the error code as a positive number.
*
     MOVE 9999 TO reply-code OF entry-reply.
     MOVE sqlcode OF sqlca TO error-code.
     MULTIPLY error-code BY -1 GIVING error-code END-MULTIPLY.

 sql-warning.
     MOVE 9998 TO reply-code OF entry-reply.
     MOVE sqlcode OF sqlca TO error-code.

 sql-notfnd.
     MOVE 9999 TO reply-code OF entry-reply.
     MOVE sqlcode OF sqlca TO error-code.
```

CALL Model: SQL Main Program

This CALL model Pathway server program modifies a single file, called PARTS in the sample database, depending on the value of the ENTRY-TYPE flag of the message received. When the flag is U, the server updates the inventory count for the specified part number. When the flag is I, it inserts a new PARTS record.

The reply message fields REPLY-CODE and ERROR-CODE are set as follows:

Reply Message	**REPLY-CODE**	**ERROR-CODE**
Operation successful	0	N.A.
Error, operation not performed	9999	Positive SQLCODE value (transformed from original negative value)
Warning, but operation successful	9998	Positive SQLCODE value
Invalid ENTRY-TYPE	9997	N.A.
CALL error occurred	9996	N.A.

The main program for the sample server illustrating the CALL model is shown in Example E-2.

Example E-2. CALL Model Main Program (page 1 of 3)

```
IDENTIFICATION DIVISION.
   PROGRAM-ID.  call-model-sql.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER.  Tandem NonStop.
   OBJECT-COMPUTER.  Tandem NonStop.
   SPECIAL-NAMES.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT msg-in
         ASSIGN TO $RECEIVE
         FILE STATUS IS receive-file-status.
       SELECT msg-out
         ASSIGN TO $RECEIVE
         FILE STATUS IS receive-file-status.
```

Example E-2. CALL Model Main Program (page 2 of 3)

```
DATA DIVISION.
 FILE SECTION.
 FD  msg-in
     LABEL RECORDS ARE OMITTED.
* The definition of ENTRY-MSG should be a COPY statement.
   01 entry-msg.
      02 pw-header.
         04 reply-code         PIC S9(4) COMP.
         04 application-code  PIC XX.
         04 function-code     PIC XX.
         04 trans-code        PIC 99.
         04 term-id           PIC X(15).
         04 log-request       PIC X.
      02 entry-type           PIC X.
      02 parts-info.
         04 partnum           PIC 9(4).
         04 partname          PIC X(18).
         04 inventory         PIC S999 COMP.
         04 location          PIC XXX.
         04 price             PIC 9(6)V99 COMP.

 FD  msg-out
     LABEL RECORDS ARE OMITTED.
     RECORD CONTAINS 1 TO 26 CHARACTERS.
* The definition of ENTRY-REPLY should be a COPY statement.
   01 entry-reply.
      02 pw-header.
         04 reply-code         PIC S9(4) COMP.
         04 filler            PIC X(22).
      02 error-code           PIC S9(4) COMP.

 WORKING-STORAGE SECTION.
   01 receive-file-status.
      02 stat-1               PIC 9.
         88 close-from-requester  VALUE 1.
      02 stat-2               PIC 9.*

* The definition of PARTS-PARAMS should be an INVOKE directive.

   01 parts-params.
      02 partnum              PIC 9(4).
      02 partname             PIC X(18).
      02 inventory            PIC S999 COMP.
      02 location             PIC XXX.
      02 price                PIC 9(6)V99 COMP.
```

Example E-2. CALL Model Main Program (page 3 of 3)

```
 PROCEDURE DIVISION.
 MAIN-SECTION SECTION.
 a-init.
     OPEN INPUT msg-in.
     OPEN OUTPUT msg-out SYNCDEPTH 1.
     PERFORM b-trans UNTIL close-from-requester.
     STOP RUN.

 b-trans.
     MOVE SPACES to entry-reply, entry-msg.
     MOVE ZERO to reply-code OF entry-reply.
     READ msg-in
          AT END STOP RUN
     END-READ.
     MOVE pw-header OF msg-in TO pw-header OF msg-out.
     IF entry-type = "U" THEN
       MOVE partnum   OF parts-info TO partnum   OF parts-
params.
       MOVE inventory OF parts-info TO inventory OF parts-
params.
       CALL update-parts USING BY REFERENCE
            parts-params
            reply-code OF entry-reply
            error-code OF entry-reply
         ON EXCEPTION MOVE 9996 TO reply-code OF entry-reply.
       END-CALL
     ELSE
     IF entry-type = "I" THEN
       MOVE partnum   OF parts-info TO partnum   OF parts-
params.
       MOVE partname  OF parts-info TO partname  OF parts-
params.
       MOVE inventory OF parts-info TO inventory OF parts-
params.
       MOVE location  OF parts-info TO location  OF parts-
params.
       MOVE price     OF parts-info TO price     OF parts-
params.
       CALL insert-parts USING BY REFERENCE
            parts-params
            reply-code OF entry-reply
            error-code OF entry-reply
         ON EXCEPTION MOVE 9996 TO reply-code OF entry-reply.
       END-CALL
     ELSE
       MOVE 9997 TO reply-code OF entry-reply
     END-IF.
     WRITE entry-reply END-WRITE.
```

CALL Model: SQL Subprograms

Following are the two subprograms that are called to perform SQL operations on the PARTS table.The reply message fields REPLY-CODE and ERROR-CODE are set as follows:

Reply Message	REPLY-CODE	ERROR-CODE
Operation successful	0	N.A.
Error, operation not performed	9999	Positive SQLCODE value (transformed from original negative value)
Warning, but operation successful	9998	Positive SQLCODE value

UPDATE Subprogram

The SQL subprogram UPDATE-PARTS that updates the PARTS table is illustrated in Example E-3.

Example E-3. SQL UPDATE Subprogram (page 1 of 2)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.  update-parts.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER.  Tandem NonStop.
   OBJECT-COMPUTER.  Tandem NonStop.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
* The definition of PARTS-RECORD should be an INVOKE directive.

   01 parts-record.
      02 partnum                PIC 9(4).
      02 partname               PIC X(18).
      02 inventory              PIC S999 COMP.
      02 location               PIC XXX.
      02 price                  PIC 9(6)V99 COMP.
     EXEC SQL  END DECLARE SECTION  END-EXEC.

     EXEC SQL  INCLUDE SQLCA  END-EXEC.

* LINKAGE SECTION.
* The definition of PARTS-PARAMS should be an INVOKE directive.

   01 parts-params.
      02 partnum                PIC 9(4).
      02 partname               PIC X(18).
      02 inventory              PIC S999 COMP.
      02 location               PIC XXX.
      02 price                  PIC 9(6)V99 COMP.
```

Example E-3. SQL UPDATE Subprogram (page 2 of 2)

```
* The definition of these LINKAGE parameters should be COPY
* statements.
   01 reply-code              PIC S9(4) COMP.
   01 error-code              PIC S9(4) COMP.

PROCEDURE DIVISION USING parts-params,
                         reply-code,
                         error-code.
 MAIN-SECTION SECTION.
 00-whenever.
     EXEC SQL  WHENEVER NOT FOUND  PERFORM :sql-notfnd   END-
EXEC.
     EXEC SQL  WHENEVER SQLERROR   PERFORM :sql-error    END-
EXEC.
     EXEC SQL  WHENEVER SQLWARNING PERFORM :sql-warning  END-
EXEC.

 start-program.
     MOVE partnum   OF parts-params TO partnum   OF parts-
record.
     MOVE inventory OF parts-params TO inventory OF parts-
record.

     EXEC SQL  UPDATE $mkt.sample.parts
               SET inventory = :parts-record.inventory
               WHERE partnum = :parts-record.partnum
     END-EXEC.
     EXIT PROGRAM.

 sql-error.
* Return the error code as a positive number.
*
     MOVE 9999 TO reply-code.
     MOVE sqlcode OF sqlca TO error-code.
     MULTIPLY error-code BY -1 GIVING error-code END-MULTIPLY.
*
 sql-warning.
     MOVE 9998 TO reply-code
     MOVE sqlcode OF sqlca TO error-code

 sql-notfnd.
     MOVE 9999 TO reply-code OF entry-reply
     MOVE sqlcode OF sqlca TO error-code.

 END PROGRAM update-parts.

?ENDUNIT
```

INSERT Subprogram

The SQL subprogram INSERT-PARTS that inserts new parts into the PARTS table is illustrated in Example E-4.

Example E-4. SQL INSERT Subprogram (page 1 of 2)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.  insert-parts.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER.  Tandem NonStop.
   OBJECT-COMPUTER.  Tandem NonStop.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.

* The definition of PARTS-RECORD should be an INVOKE directive.

   01 parts-record.
      02 partnum                PIC 9(4).
      02 partname               PIC X(18).
      02 inventory              PIC S999 COMP.
      02 location               PIC XXX.
      02 price                  PIC 9(6)V99 COMP.

     EXEC SQL  END DECLARE SECTION  END-EXEC.

     EXEC SQL  INCLUDE SQLCA  END-EXEC.

 LINKAGE SECTION.
* The definition of PARTS-PARAMS should be an INVOKE directive.

   01 parts-params.
      02 partnum                PIC 9(4).
      02 partname               PIC X(18).
      02 inventory              PIC S999 COMP.
      02 location               PIC XXX.
      02 price                  PIC 9(6)V99 COMP.
*
* The definition of these LINKAGE parameters should be COPY
* statements.
   01 reply-code                PIC S9(4) COMP.
   01 error-code                PIC S9(4) COMP.
```

Example E-4. SQL INSERT Subprogram (page 2 of 2)

```
 PROCEDURE DIVISION USING parts-params,
                          reply-code,
                          error-code.

 MAIN-SECTION SECTION.
 00-whenever.
     EXEC SQL  WHENEVER NOT FOUND CONTINUE               END-
EXEC.
     EXEC SQL  WHENEVER SQLERROR   PERFORM :sql-error    END-
EXEC.
     EXEC SQL  WHENEVER SQLWARNING PERFORM :sql-warning  END-
EXEC.

 start-program.
     MOVE partnum   OF parts-params TO partnum   OF parts-
record.
     MOVE partname  OF parts-params TO partname  OF parts-
record.
     MOVE inventory OF parts-params TO inventory OF parts-
record.
     MOVE location  OF parts-params TO location  OF parts-
record.
     MOVE price     OF parts-params TO price     OF parts-
record.

     EXEC SQL  INSERT INTO $mkt.sample.parts
               VALUES ( :parts-record.partnum,
                        :parts-record.partname,
                        :parts-record.inventory,
                        :parts-record.location,
                        :parts-record.price )
     END-EXEC.
     EXIT PROGRAM.

 sql-error.
* Return the error code as a positive number.
     MOVE 9999 TO reply-code.
     MOVE sqlcode OF sqlca TO error-code.
     MULTIPLY error-code BY -1 GIVING error-code END-MULTIPLY.

 sql-warning.
     MOVE 9998 TO reply-code
     MOVE sqlcode OF sqlca TO error-code

 END PROGRAM insert-parts.
```

Index