

# HP NonStop SQL/MP Programming Manual for C

## Abstract

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

## 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, G06.00 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
429847-008	August 2012

## Document History

Part Number	Product Version	Published
429847-002	NonStop SQL/MP G06	December 2003
429847-003	NonStop SQL/MP G06	December 2004
429847-004	NonStop SQL/MP G06	April 2005
429847-005	NonStop SQL/MP G06	February 2006
429847-007	NonStop SQL/MP G06 and H01	August 2010
429847-008	NonStop SQL/MP G06 and H01	August 2012

---

---

---

---

---

# Legal Notices

© Copyright 2012 Hewlett-Packard Development Company, L.P.

## Legal Notice

Confidential computer software. Valid license from HP required for possession, use or copying.  
Consistent with FAR 12.211 and 12.212, Commercial

Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks, and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. The OSF documentation and the OSF software to which it relates are derived in part from materials supplied by the following: © 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation.

© 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991

Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation. OSF software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.



# HP NonStop SQL/MP Programming Manual for C

[Index](#)[Examples](#)[Figures](#)[Tables](#)

## [Legal Notices](#)

[What's New in This Manual](#) xi[Manual Information](#) xi[New and Changed Information](#) xi[About This Manual](#) xv[Who Should Read This Guide](#) xv[Related Manuals](#) xv[Notation Conventions](#) xx[HP Encourages Your Comments](#) xxii

## [1. Introduction](#)

[Advantages of Using Embedded SQL Statements](#) 1-1[Developing a C Program](#) 1-1[Declaring and Using Host Variables](#) 1-2[Embedding SQL/MP Statements and Directives](#) 1-3[Calling SQL/MP System Procedures](#) 1-4[Compiling and Executing a Host-Language Program](#) 1-5[Processing Errors, Warnings, and Status Information](#) 1-5[Dynamic SQL](#) 1-6[SQL/MP Version Management](#) 1-7

## [2. Host Variables](#)

[Specifying a Declare Section](#) 2-1[Coding Host Variable Names](#) 2-2[Using Corresponding SQL and C Data Types](#) 2-3[Specifying Host Variables in SQL Statements](#) 2-6[Declaring and Using Host Variables](#) 2-7[Fixed-Length Character Data](#) 2-7[Variable-Length Character Data](#) 2-9[Structures](#) 2-9[Decimal Data Types](#) 2-11

## **2. Host Variables (continued)**

- [Fixed-Point Data Types](#) 2-11
- [Date-Time and INTERVAL Data Types](#) 2-13
- [Using Indicator Variables for Null Values](#) 2-17
  - [Inserting a Null Value](#) 2-17
  - [Testing For a Null Value](#) 2-17
  - [Retrieving Rows With Null Values](#) 2-18
- [Creating Host Variables Using the INVOKE Directive](#) 2-18
  - [Advantages of Using an INVOKE Directive](#) 2-19
  - [C Structures Generated by the INVOKE Directive](#) 2-19
  - [Using Indicator Variables With the INVOKE Directive](#) 2-22
  - [Using INVOKE With SQLCI](#) 2-24
- [Associating a Character Set With a Host Variable](#) 2-24
  - [Treatment in C Statements](#) 2-25
  - [VARCHAR Data Type](#) 2-25

## **3. SQL/MP Statements and Directives**

- [Embedding SQL Statements](#) 3-1
  - [Coding Statements and Directives](#) 3-1
  - [Placing Statements and Directives](#) 3-2
- [Finding Information](#) 3-3

## **4. Data Retrieval and Modification**

- [Opening and Closing Tables and Views](#) 4-2
  - [Causes of SQL Error 8204 \(Lost Open Error\)](#) 4-2
  - [Recovering From SQL Error 8204](#) 4-3
- [Single-Row SELECT Statement](#) 4-4
  - [Using a Column Value to Select Data](#) 4-5
  - [Using a Primary Key Value to Select Data](#) 4-6
- [Multirow SELECT Statement](#) 4-6
  - [Simple Example](#) 4-7
  - [A More Complex Example](#) 4-7
  - [The Most Complex Example](#) 4-7
- [INSERT Statement](#) 4-8
  - [Inserting a Single Row](#) 4-9
  - [Inserting a Null Value](#) 4-9
  - [Inserting a Timestamp Value](#) 4-10
- [UPDATE Statement](#) 4-10
  - [Updating a Single Row](#) 4-11

## **4. Data Retrieval and Modification (continued)**

- [Updating Multiple Rows](#) 4-12
- [Updating Columns With Null Values](#) 4-12
- [DELETE Statement](#) 4-12
  - [Deleting a Single Row](#) 4-13
  - [Deleting Multiple Rows](#) 4-13
- [Using SQL Cursors](#) 4-14
  - [Steps for Using a Cursor](#) 4-15
  - [Process Access ID \(PAID\) Requirements](#) 4-16
  - [Cursor Position](#) 4-16
  - [Cursor Stability](#) 4-17
  - [Virtual Sequential Block Buffering \(VSBB\)](#) 4-17
  - [DECLARE CURSOR Statement](#) 4-18
  - [OPEN Statement](#) 4-19
  - [FETCH Statement](#) 4-20
  - [Multirow SELECT Statement](#) 4-21
  - [UPDATE Statement](#) 4-22
  - [Multirow DELETE Statement](#) 4-23
  - [CLOSE Statement](#) 4-24
  - [Using Foreign Cursors](#) 4-24

## **5. SQL/MP System Procedures**

- [Guardian System Procedures](#) 5-2
- [cextdecs Header File](#) 5-2
- [SQL Message File](#) 5-2
- [SQLCADISPLAY](#) 5-3
- [SQLCAFSCODE](#) 5-8
- [SQLCAGETINFOLIST](#) 5-9
- [SQLCATOBUFFER](#) 5-14
- [SQLGETCATALOGVERSION](#) 5-18
- [SQLGETOBJECTVERSION](#) 5-19
- [SQLGETSYSTEMVERSION](#) 5-19
- [SQLSADISPLAY](#) 5-20

## **6. Explicit Program Compilation**

- [Explicit Program Compilation](#) 6-1
- [Developing a C Program in the Guardian Environment](#) 6-5
  - [Using TACL DEFINES in the Guardian Environment](#) 6-6
  - [Specifying the SQL Pragma in the Guardian Environment](#) 6-7

## **6. Explicit Program Compilation (continued)**

<a href="#">Running the TNS C Compiler in the Guardian Environment</a>	6-9
<a href="#">Running the TNS/R NMC and TNS/E CCOMP Compiler in the Guardian Environment</a>	6-10
<a href="#">Binding SQL Program Files in the Guardian Environment</a>	6-11
<a href="#">Running the SQL Compiler in the Guardian Environment</a>	6-12
<a href="#">SQL Program File Format</a>	6-24
<a href="#">SQL Compiler Listings</a>	6-25
<a href="#">Developing a C Program in the OSS Environment</a>	6-28
<a href="#">Using TACL DEFINES in the OSS Environment</a>	6-29
<a href="#">Using the c89 Utility in the OSS Environment</a>	6-30
<a href="#">Developing a C Program in a PC Host Environment</a>	6-33
<a href="#">Using CONTROL Directives</a>	6-34
<a href="#">Static SQL Statements</a>	6-34
<a href="#">Dynamic SQL Statements</a>	6-36
<a href="#">Using Compatible Compilation Tools</a>	6-36
<a href="#">C Compiler</a>	6-36
<a href="#">SQL Compiler</a>	6-36
<a href="#">SQL Program Files</a>	6-37

## **7. Program Execution**

<a href="#">Required Access Authority</a>	7-1
<a href="#">Using TACL DEFINES</a>	7-2
<a href="#">Entering the TACL RUN Command</a>	7-3
<a href="#">Running a Program in the OSS Environment</a>	7-3
<a href="#">Running a Program at a Low PIN</a>	7-4
<a href="#">Interactive Commands</a>	7-5
<a href="#">Programmatic Commands</a>	7-5
<a href="#">Pathway Environment</a>	7-6
<a href="#">Determining Compatibility With the SQL Executor</a>	7-7

## **8. Program Invalidation and Automatic SQL Recompilation**

<a href="#">Program Invalidation</a>	8-1
<a href="#">SQL Compiler Validation Functions</a>	8-1
<a href="#">Causes of Program Invalidation</a>	8-2
<a href="#">File-Label and Catalog Inconsistencies</a>	8-4
<a href="#">Preventing Program Invalidation</a>	8-4
<a href="#">Automatic SQL Recompilation</a>	8-5
<a href="#">Causes of Automatic Recompilation</a>	8-6



## **8. Program Invalidation and Automatic SQL Recompilation (continued)**

- [Run-Time Recompilation Errors](#) 8-9
- [Preventing Automatic Recompilations](#) 8-9

## **9. Error and Status Reporting**

- [Using the INCLUDE STRUCTURES Directive](#) 9-1
- [Generating Structures With Different Versions](#) 9-3
- [Checking the Version of the C Compiler](#) 9-3
- [Sharing Structures](#) 9-3
- [Returning Error and Warning Information](#) 9-4
  - [Checking the sqlcode Variable](#) 9-4
  - [Using the WHENEVER Directive](#) 9-6
  - [Returning Information From the SQLCA Structure](#) 9-12
- [Returning Performance and Statistics Information](#) 9-13
  - [Declaring the SQLSA Structure](#) 9-13
  - [Using the SQLSA Structure](#) 9-13

## **10. Dynamic SQL Operations**

- [Uses for Dynamic SQL](#) 10-1
- [Dynamic SQL Statements](#) 10-2
- [Dynamic SQL Features](#) 10-3
  - [SQLDA Structure, Names Buffer, and Collation Buffer](#) 10-3
  - [Input Parameters and Output Variables](#) 10-11
  - [Null Values](#) 10-16
  - [Dynamic Allocation of Memory](#) 10-18
  - [Using Dynamic SQL Cursors](#) 10-20
- [Developing a Dynamic SQL Program](#) 10-23
  - [Specify the SQL Pragma](#) 10-23
  - [Copy any External Declarations](#) 10-23
  - [Declare the sqlcode Variable and Host Variables](#) 10-23
  - [Specify Any WHENEVER Directives](#) 10-23
  - [Specify the INCLUDE STRUCTURES Directive](#) 10-24
  - [Declare the SQLDA Structure and Names Buffer](#) 10-24
  - [Declare an SQLSA Structure](#) 10-24
  - [Process the Input Parameters](#) 10-24
  - [Read and Compile the SQL Statement](#) 10-25
  - [Process the Output Variables](#) 10-25
  - [Perform the Database Request and Display the Values](#) 10-27

## **10. Dynamic SQL Operations (continued)**

<a href="#"><u>Allocate Memory for the SQLDA Structures and Names Buffers</u></a>	10-29
<a href="#"><u>Allocate and Fill In Output Variables</u></a>	10-33
<a href="#"><u>Developing a Dynamic SQL Pathway Server</u></a>	10-36
<a href="#"><u>Dynamic SQL Sample Programs</u></a>	10-37
<a href="#"><u>Basic Dynamic SQL Program</u></a>	10-37
<a href="#"><u>Detailed Dynamic SQL Program</u></a>	10-42

## **11. Character Processing Rules (CPRL) Procedures**

<a href="#"><u>cextdecs Header File</u></a>	11-2
<a href="#"><u>CPRL Return Codes</u></a>	11-2
<a href="#"><u>CPRL_ARE</u></a>	11-3
<a href="#"><u>CPRL_AREALPHAS</u></a>	11-4
<a href="#"><u>CPRL_ARENUMERICS</u></a>	11-5
<a href="#"><u>CPRL_COMPARE1ENCODED</u></a>	11-6
<a href="#"><u>CPRL_COMPARE</u></a>	11-7
<a href="#"><u>CPRL_COMPAREOBJECTS</u></a>	11-8
<a href="#"><u>CPRL_DECODE</u></a>	11-9
<a href="#"><u>CPRL_DOWNSHIFT</u></a>	11-10
<a href="#"><u>CPRL_ENCODE</u></a>	11-11
<a href="#"><u>CPRL_GETALPHATABLE</u></a>	11-12
<a href="#"><u>CPRL_GETCHARCLASSTABLE</u></a>	11-13
<a href="#"><u>CPRL_GETDOWNSHIFTTABLE</u></a>	11-14
<a href="#"><u>CPRL_GETFIRST</u></a>	11-15
<a href="#"><u>CPRL_GETLAST</u></a>	11-16
<a href="#"><u>CPRL_GETNEXTINSEQUENCE</u></a>	11-17
<a href="#"><u>CPRL_GETNUMTABLE</u></a>	11-18
<a href="#"><u>CPRL_GETSPECIALTABLE</u></a>	11-19
<a href="#"><u>CPRL_GETUPSHIFTTABLE</u></a>	11-20
<a href="#"><u>CPRL_INFO</u></a>	11-20
<a href="#"><u>CPRL_READOBJECT</u></a>	11-22
<a href="#"><u>CPRL_UPSHIFT</u></a>	11-23

### **A. SQL/MP Sample Database**

### **B. Memory Considerations**

<a href="#"><u>SQL/MP Internal Structures</u></a>	B-1
<a href="#"><u>Using the SQLMEM Pragma</u></a>	B-2
<a href="#"><u>Estimating Memory Requirements</u></a>	B-2
<a href="#"><u>Avoiding Memory Stack Overflows</u></a>	B-4

## **C. Maximizing Local Autonomy**

- [Using a Local Partition](#) C-1
- [Using TACL DEFINES](#) C-2
- [Using Current Statistics](#) C-2
- [Skipping Unavailable Partitions](#) C-3

## **D. Converting C Programs**

- [Generating SQL Data Structures](#) D-1
- [Generating SQLDA Structures](#) D-2
  - [Generating a Version 300 \(or Later\) SQLDA Structure](#) D-3
  - [Generating a Version 2 SQLDA Structure](#) D-3
  - [Generating a Version 1 SQLDA Structure](#) D-6
- [Planning for Future PVUs](#) D-8
  - [SQL/MP Version Procedures](#) D-8
  - [RELEASE1 and RELEASE2 Options](#) D-8

## **Index**

## **Examples**

- [Example 1-1. Static SQL Statements in a C Program](#) 1-4
- [Example 1-2. Dynamic SQL Statements in a C Program](#) 1-6
- [Example 2-1. Creating Valid DATETIME and INTERVAL Data Types](#) 2-16
- [Example 2-2. CREATE TABLE Statements](#) 2-20
- [Example 2-3. Structures Generated by the INVOKE Directive](#) 2-21
- [Example 4-1. Using a Static SQL Cursor in a C Program](#) 4-14
- [Example 5-1. Example of the SQLCAGETINFOLIST Procedure](#) 5-13
- [Example 5-2. Example of the SQLSADISPLAY Display](#) 5-22
- [Example 6-1. Sample SQL Compiler Listing](#) 6-25
- [Example 9-1. Checking the sqlcode Variable](#) 9-5
- [Example 9-2. Enabling and Disabling the WHENEVER Directive](#) 9-9
- [Example 9-3. Using the WHENEVER Directive](#) 9-10
- [Example 9-4. Version 300-325 SQLSA Structure](#) 9-15
- [Example 9-5. Version 330 \(or later\) SQLSA Structure](#) 9-16
- [Example 10-1. SQLDA Structure and Buffers](#) 10-7
- [Example 10-2. Getting Parameter Values](#) 10-15
- [Example 10-3. Using Statement and Cursor Host Variables](#) 10-22
- [Example 10-4. Allocating the SQLDA Structure](#) 10-30
- [Example 10-5. Allocating Memory for Parameters and Columns](#) 10-32
- [Example 10-6. Displaying Output](#) 10-34
- [Example 10-7. Basic Dynamic SQL Program](#) 10-39

## Examples (continued)

<a href="#">Example 10-8.</a>	<a href="#">Detailed Dynamic SQL Program</a>	10-44
<a href="#">Example A-1.</a>	<a href="#">COPYLIB File for Sample Database</a>	A-3
<a href="#">Example D-1.</a>	<a href="#">Version 2 SQLDA Structure</a>	D-4
<a href="#">Example D-2.</a>	<a href="#">Version 1 SQLDA Structure</a>	D-6

## Figures

<a href="#">Figure i.</a>	<a href="#">NonStop SQL/MP Library</a>	xvii
<a href="#">Figure ii.</a>	<a href="#">Program Development, System and OSS Manuals</a>	xviii
<a href="#">Figure 6-1.</a>	<a href="#">Explicit SQL Compilation of a C Program on TNS</a>	6-3
<a href="#">Figure 6-2.</a>	<a href="#">Explicit SQL Compilation of a C Program on TNS/R</a>	6-4
<a href="#">Figure 6-3.</a>	<a href="#">Explicit SQL Compilation of a C Program on TNS/E</a>	6-5
<a href="#">Figure 6-4.</a>	<a href="#">SQL/MP Program File Format</a>	6-24
<a href="#">Figure 7-1.</a>	<a href="#">Processes Running on a NonStop System</a>	7-4
<a href="#">Figure 8-1.</a>	<a href="#">Timestamp Check</a>	8-8
<a href="#">Figure 10-1.</a>	<a href="#">DESCRIBE INPUT's Effect on Names Buffer</a>	10-18
<a href="#">Figure A-1.</a>	<a href="#">SQL/MP Sample Database Relations</a>	A-2

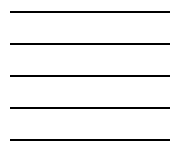
## Tables

<a href="#">Table i.</a>	<a href="#">NonStop SQL/MP Library</a>	xvi
<a href="#">Table ii.</a>	<a href="#">Program Development Manuals</a>	xix
<a href="#">Table iii.</a>	<a href="#">Guardian Manuals</a>	xx
<a href="#">Table iv.</a>	<a href="#">Open System Services (OSS) Manuals</a>	xx
<a href="#">Table 1-1.</a>	<a href="#">SQL/MP Statements and Directives</a>	1-3
<a href="#">Table 2-1.</a>	<a href="#">Corresponding SQL and C Character Data Types</a>	2-3
<a href="#">Table 2-2.</a>	<a href="#">Corresponding SQL and C Numeric, Date-Time, and INTERVAL Data Types</a>	2-4
<a href="#">Table 2-3.</a>	<a href="#">Date-Time and INTERVAL Data Types</a>	2-13
<a href="#">Table 3-1.</a>	<a href="#">Summary of SQL/MP Statements and Directives</a>	3-3
<a href="#">Table 3-2.</a>	<a href="#">C Compiler Pragmas for SQL/MP</a>	3-7
<a href="#">Table 4-1.</a>	<a href="#">SQL/MP Statements for Data Retrieval and Modification</a>	4-1
<a href="#">Table 4-2.</a>	<a href="#">Determining the Cursor Position</a>	4-16
<a href="#">Table 5-1.</a>	<a href="#">SQL/MP System Procedures</a>	5-1
<a href="#">Table 5-2.</a>	<a href="#">Guardian System Procedures That Return SQL Information</a>	5-2
<a href="#">Table 5-3.</a>	<a href="#">SQLCAGETINFOLIST Procedure Error Codes</a>	5-11
<a href="#">Table 5-4.</a>	<a href="#">SQLCAGETINFOLIST Procedure Item Codes</a>	5-11
<a href="#">Table 5-5.</a>	<a href="#">SQLSADISPLAY Procedure Display Elements</a>	5-22
<a href="#">Table 6-1.</a>	<a href="#">C Compilers</a>	6-2
<a href="#">Table 6-2.</a>	<a href="#">Compilation Mode and Execution Environment</a>	6-2

**Tables (continued)**

<a href="#">Table 9-1.</a>	<a href="#">C Compiler Pseudocode for Checking the sqlcode Variable</a>	9-6
<a href="#">Table 9-2.</a>	<a href="#">C Identifiers Generated by the INCLUDE SQLCA Directive</a>	9-12
<a href="#">Table 9-3.</a>	<a href="#">System Procedures for the SQLCA Structure</a>	9-12
<a href="#">Table 9-4.</a>	<a href="#">C Identifiers Generated by the INCLUDE SQLSA Directive</a>	9-14
<a href="#">Table 9-5.</a>	<a href="#">SQLSA Structure Fields</a>	9-17
<a href="#">Table 10-1.</a>	<a href="#">Dynamic SQL Statements</a>	10-2
<a href="#">Table 10-2.</a>	<a href="#">C Identifiers Generated by the INCLUDE SQLDA Directive</a>	10-5
<a href="#">Table 10-3.</a>	<a href="#">SQLDA Structure Fields</a>	10-5
<a href="#">Table 10-4.</a>	<a href="#">SQLDA Data Type Declarations</a>	10-8
<a href="#">Table 10-5.</a>	<a href="#">SQLDA Date-Time and INTERVAL Declarations</a>	10-10
<a href="#">Table 10-6.</a>	<a href="#">SQLDA Character-Set IDs</a>	10-11
<a href="#">Table 11-1.</a>	<a href="#">Character Processing Rules (CPRL) Procedures</a>	11-1
<a href="#">Table B-1.</a>	<a href="#">SQL/MP Data Structures</a>	B-1
<a href="#">Table B-2.</a>	<a href="#">Virtual Memory Requirements for SQL Statements</a>	B-3
<a href="#">Table D-1.</a>	<a href="#">Changes to SQL Data Structures</a>	D-2
<a href="#">Table D-2.</a>	<a href="#">Version 2 SQLDA Structure Fields</a>	D-4
<a href="#">Table D-3.</a>	<a href="#">Version 1 SQLDA Structure Fields</a>	D-6





# What's New in This Manual

## Manual Information

### Abstract

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

### 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, G06.00 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
429847-008	August 2012

### Document History

Part Number	Product Version	Published
429847-002	NonStop SQL/MP G06	December 2003
429847-003	NonStop SQL/MP G06	December 2004
429847-004	NonStop SQL/MP G06	April 2005
429847-005	NonStop SQL/MP G06	February 2006
429847-007	NonStop SQL/MP G06 and H01	August 2010
429847-008	NonStop SQL/MP G06 and H01	August 2012

## New and Changed Information

### Changes to the H06.25/J06.14 manual:

- Added -Wsqlconnect compiler option in [-Wsqlconnect](#) on page 6-33.
- Added -HP\_NSK\_CONNECT\_MODE environment variable option in [HP\\_NSK\\_CONNECT\\_MODE](#) on page 6-34.

## Changes to the H06.21/J06.06 manual

- Updated footnote about compiler version support under [Table 6-1, C Compilers](#), on page 6-2.

## Changes to the G06.28 Manual

- Added a [Note](#) on page 2-5 about the nonsupport for unsigned long long data type.
- Changed the format of `short output_file_number` under SQLCADISPLAY on pages [5-4](#) and [5-20](#).
- Updated the information in [Section 6, Explicit Program Compilation](#) with the information from the *SQL Supplement for H-series RVUs*.
- Corrected [Example 9-5](#) on page 9-16.

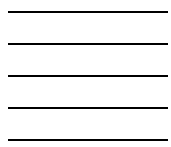


- Corrected two field names in:
  - [Table 9-5](#) on page 9-17
  - [Table D-1](#) on page D-2

## Changes in the G06.26 Manual

- Updated information related to process access:
  - On page [2-19](#), for an INVOKE directive
  - On page [4-4](#), for a SELECT statement
  - On page [4-8](#), for an INSERT statement
  - On page [4-10](#), for an UPDATE statement
  - On page [4-13](#), for a DELETE statement
  - On page [4-16](#), for an SQL cursor
  - On page [4-19](#), for an OPEN CURSOR statement
  - On page [4-20](#), for a FETCH statement
  - On page [4-21](#), for a multirow SELECT statement
  - On page [4-22](#), for an UPDATE statement with a cursor
  - On page [4-23](#), for a DELETE statement with a cursor
  - On page [6-21](#), for an UPDATE STATISTICS statement
- Added information about compiling NonStop C programs in the PC environment under [Developing a C Program in a PC Host Environment](#) on page 6-33.
- Added information about process access privileges under [Required Access Authority](#) on page 7-1.
- Corrected coding error on page [10-50](#) by shifting code from line 374 to 372.
- Changed the real memory from 2 KB to 16 KB pages under Estimating Memory Requirements on page [B-4](#).





# About This Manual

This manual describes the NonStop SQL/MP programmatic interface for the HP implementation of the C language. Using this interface, a C program can access a NonStop SQL/MP database using embedded SQL statements and directives.

## Who Should Read This Guide

This manual is intended for application programmers who are embedding SQL statements and directives in a C program. The reader should be familiar with:

- The C programming language
- NonStop SQL/MP terms and concepts as described in the *Introduction to NonStop SQL/MP*
- The HP NonStop operating system, including either the Guardian or HP NonStop Open System Services (OSS) environment

## Related Manuals

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

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

[Table i](#) describes the manuals in the HP NonStop SQL/MP library.

---

**Table i. NonStop SQL/MP Library**

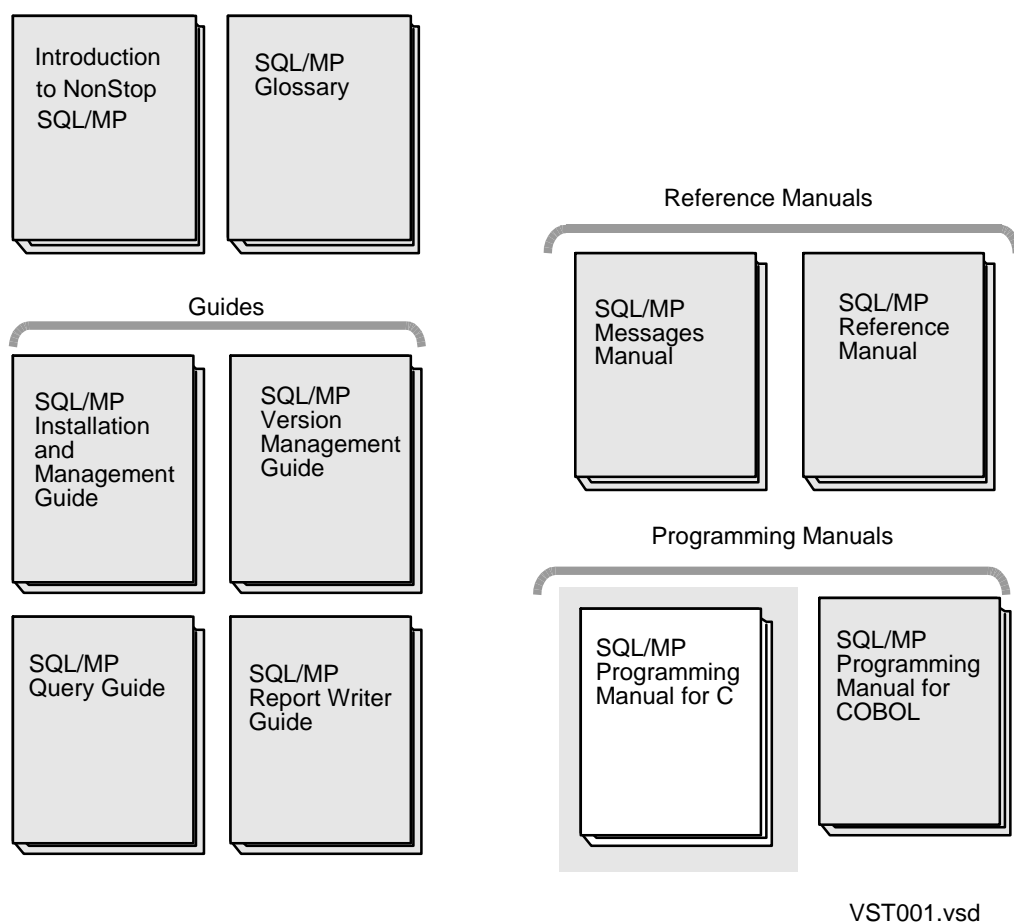
<b>Manual</b>	<b>Description</b>
<i>Introduction to NonStop SQL/MP</i>	Introduces the NonStop SQL/MP relational database management system.
<i>SQL/MP Reference Manual</i>	Describes the NonStop SQL/MP language elements, including expressions, functions, commands, statements, SQLCI utilities and commands, and report writer commands. This manual is the printed version of Online Help.
<i>SQL/MP Messages Manual</i>	Describes error and warning numbers and messages returned by NonStop SQL, the SQL file system, and FastSort.
<i>SQL/MP Query Guide</i>	Describes how to retrieve and modify data in a NonStop SQL/MP database and how to analyze and improve query performance.
<i>SQL/MP Version Management Guide</i>	Describes the rules governing version management for the NonStop SQL/MP software, catalogs, objects, messages, programs, and data structures.
<i>SQL/MP Installation and Management Guide</i>	Describes how to plan, install, create, and manage a NonStop SQL/MP database and SQL programs.
<i>SQL/MP Report Writer Guide</i>	Describes how to use report writer commands and SQLCI options to design and produce reports.
<i>SQL/MP Programming Manual for C</i> <i>SQL/MP Programming Manual for COBOL</i>	Describes the NonStop SQL/MP programmatic interface for C and COBOL applications.

---

[Figure i](#) shows the manuals in the NonStop SQL/MP library.

---

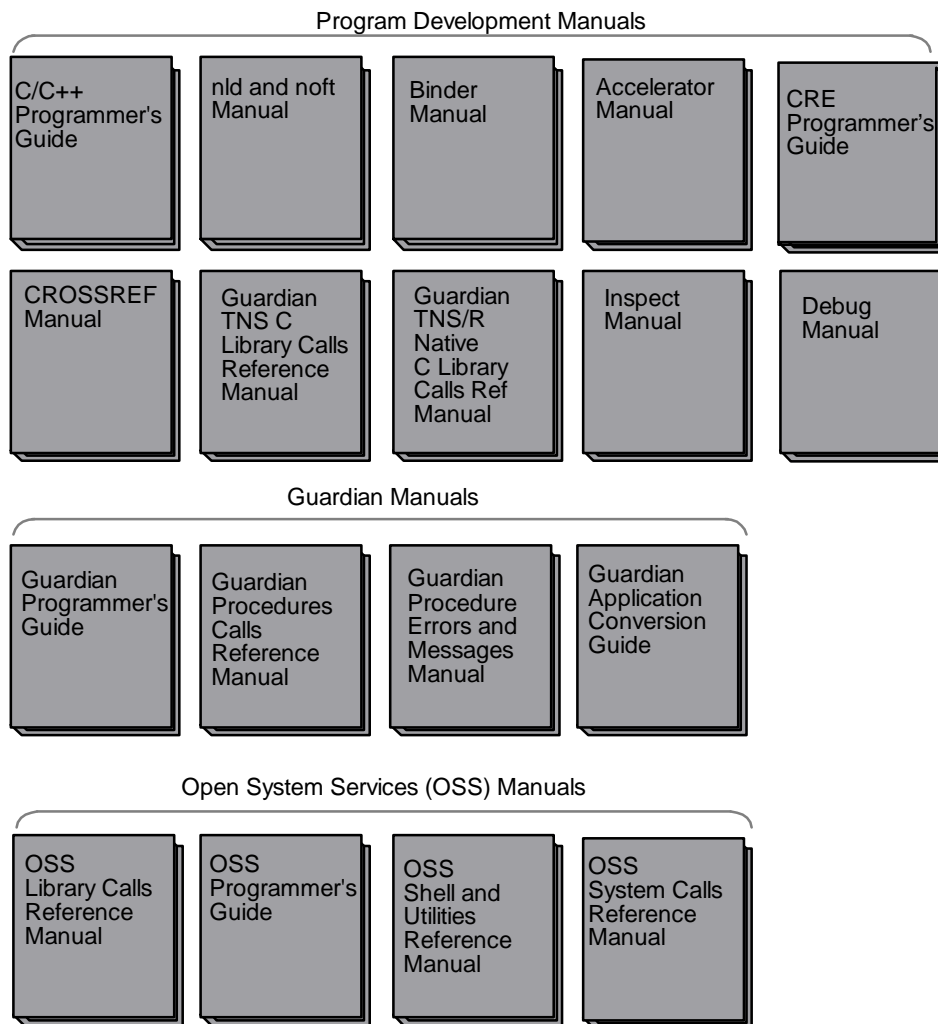
**Figure i. NonStop SQL/MP Library**



In addition to the NonStop SQL/MP library, program development, Guardian, and OSS manuals can be useful to a C programmer. They are shown in [Figure ii](#) and described in [Table ii](#), [Table iii](#) on page xx, and [Table iv](#) on page xx.

---

**Figure ii. Program Development, System and OSS Manuals**



VST011.vsd

**Table ii. Program Development Manuals**

<b>Manual</b>	<b>Description</b>
<i>C /C++ Programmer's Guide</i>	Describes HP extensions to the C and C++ languages, including how to write applications that run in either the Guardian or OSS environments.
<i>nld and noft Manual</i>	Describes how to use the native link editor ( <code>nld</code> ) and the native object file tool ( <code>noft</code> ).
<i>Binder Manual</i>	Describes the Binder program, an interactive linker that enables you to examine, modify, and combine object files and to generate load maps and cross-reference listings.
<i>Accelerator Manual</i>	Describes how to use the Accelerator to optimize TNS object code for the TNS/R execution environment.
<i>CRE Programmer's Guide</i>	Describes the Common Run-Time Environment (CRE) and how to write and run mixed-language programs.
<i>CROSSREF Manual</i>	Describes the CROSSREF program, which produces a cross-reference listing of selected identifiers in an application.
<i>Guardian TNS C Library Calls Reference Manual</i>	Describes the C run-time library available to TNS and accelerated programs in the Guardian environment.
<i>Guardian TNS/R Native C Library Calls Reference Manual</i>	Describes the C run-time library available to TNS/R programs in the Guardian environment.
<i>Inspect Manual</i>	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.
<i>Debug Manual</i>	Describes the Debug program, an interactive machine-level debugger.

**Table iii. Guardian Manuals**

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

**Table iv. Open System Services (OSS) Manuals**

Manual	Description
<i>Open System Services Library Calls Reference Manual</i>	Describes the syntax and semantics of the C run-time library in the OSS environment.
<i>Open System Services Programmer's Guide</i>	Describes how to use the OSS application programming interface to the operating system.
<i>Open System Services Shell and Utilities Reference Manual</i>	Describes the syntax and semantics for using the OSS shell and utilities.
<i>Open System Services System Calls Reference Manual</i>	Describes the syntax and programming considerations for using OSS system calls.

## Notation Conventions

### General Syntax Notation

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

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

```
EXEC SQL CONTROL EXECUTOR PARALLEL EXECUTION ON;
```

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

```
#pragma SQL [ option ]
            [ ( option [ , option ]... ) ]
```



**Computer type.** Computer type letters within text indicate C and Open System Services (OSS) keywords and reserved words; enter these items exactly as shown. For example:

```
SYSTYPEpOSS
```

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

```
OUT [ list-file ]
```

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:

```
[ , PAGES num-pages      ]
[ , SQLMAP                ]
[ , WHENEVERLIST          ]
[ , RELEASE1 | RELEASE2   ]
```

**{ } 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:

```
{ PAGE [S]      }
{ BYTE [S]      }
{ MEGABYTE [S]  }
```

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

```
[ RECOMPILEONDEMAND | RECOMPILEALL ]
```

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

```
[ , run-option ]...
```

An ellipsis in a programming example indicates that one or more lines of source code have been omitted.

```
#include <cextdecs(SQLCAFS CODE)>
...
short fserr;
EXEC SQL INCLUDE SQLCA;
...
fserr = SQLCAFS CODE ((short *) &sqlca);
...
```

**Punctuation.** Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown.

**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:

```
SQLCOMP / IN object-file [ , OUT [ list-file ] ] /
```

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

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

**i and o.** In the syntax diagrams for system procedure calls, *i* and *o* are used as follows:

```
/* i    */    Input parameter—passes data to the procedure
/* o    */    Output parameter—returns data to the calling program
/* i:o  */    Input and output parameter—both passes and returns data
```

An example of the syntax for a procedure call is as follows:

```
#include <cextdecs(SQLCAFSCODE)>

short SQLCAFSCODE (  short *sqlca,                /* i */
                    [ short first_flg ] );        /* i */
```

## 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](mailto: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 NonStop SQL/MP for C programs.

This section discusses:

- [Advantages of Using Embedded SQL Statements](#)
- [Developing a C Program](#)
- [Dynamic SQL](#) on page 1-6
- [SQL/MP Version Management](#) on page 1-7

## Advantages of Using Embedded SQL Statements

Using embedded SQL statements and directives in a C program to access an SQL/MP database has these advantages:

- A high-level, efficient database language—You code a request to access the database using SQL statements. The SQL/MP optimizer then generates an efficient path 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.
- Use of C statements to process data—You can access the database using SQL statements and then use C 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.

## Developing a C Program

You can embed static or dynamic SQL statements in a C 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. For a dynamic SQL statement, you code a placeholder variable for the statement, and then construct, SQL compile, and run the statement at run time.

## Declaring and Using Host Variables

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

You declare host variables in a Declare Section in the variable declarations part of your program. A Declare Section begins with the BEGIN DECLARE SECTION directive and ends with the END DECLARE SECTION directive. In this example, `host_variable1`, `host_variable2`, `number` and `name` are host variables.

```
EXEC SQL BEGIN DECLARE SECTION;
int host_variable1;           /* int host variable */
char host_variable2[19];      /* char host variable */
struct host_variable_names
{
    long number;              /* long host variable */
    char name[31];            /* char host variable */
} hv_names;
...
EXEC SQL END DECLARE SECTION;
```

The C compiler accepts the CHARACTER SET clause in a host-variable declaration to associate a single-byte or double-byte character set such as Kanji or KSC5601 with a host variable.

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

```
EXEC SQL SELECT column1 INTO :host_variable1 FROM =table
      WHERE column1 > :host_variable2;
strcpy(new_name, host_variable1);
```

For more information, see [Section 2, Host Variables](#).

## Embedding SQL/MP Statements and Directives

[Table 1-1](#) lists the SQL/MP statements and directives you can embed in a C 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

---

Precede an embedded SQL statement or directive with the EXEC SQL keywords and terminate it with a semicolon (;).

[Example 1-1](#) shows an example of static SQL statements embedded in a C program:

---

### Example 1-1. Static SQL Statements in a C Program

```
/* C variable declarations */
...
EXEC SQL BEGIN DECLARE SECTION;

struct in_parts_struct      /* host variables */
{
    short in_partnum;
    long  in_price;
    char  in_partdesc[19];
} in_parts;
EXEC SQL END DECLARE SECTION;

void insert_function(void)
{
    ...
    in_parts.in_partnum = 4120;
    in_parts.in_price = 6000000;
    strcpy (in_parts.in_partdesc, "V8 DISK OPTION    ");
    EXEC SQL
        INSERT INTO $vol5.sales.parts (partnum, price, partdesc)
            VALUES (:in_parts.in_partnum,
                SETSCALE (:in_parts.in_price,2), /* scale is 2. */
                :in_parts.in_partdesc);
    ... }

```

---

For more information, see [Section 3, SQL/MP Statements and Directives](#) and [Section 4, Data Retrieval and Modification](#).

## Calling SQL/MP System Procedures

NonStop SQL/MP provides system procedures, written in TAL, that perform various SQL operations and functions. For example, the SQLCADISPLAY procedure returns error information from the SQLDA structure after an SQL statement runs.

You call SQL system procedures from a C program in the same manner you call other system procedures (for example, FILE\_OPEN\_, FILE\_CLOSE\_, or WRITEREAD). The cextdecs header file contains source declarations for these procedures that you can include in a program. This example calls the SQLCADISPLAY procedure by using all default parameters:

```
#include <cextdecs(SQLCADISPLAY)>
...
SQLCADISPLAY( (short *) &sqlca);
... /* Process information from the SQLCA structure */

```

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

## Compiling and Executing a Host-Language Program

The steps to compile and run a C program that contains embedded SQL statements are similar to the steps you follow for a C program that does not contain embedded SQL statements. You must perform only one extra step for a host-language program: compiling the embedded SQL statements using the SQL compiler.

1. Compile the C source file (or files) that contain the embedded SQL statements using the C compiler. The C compiler generates an object file that contains C object code and SQL source statements.
2. If necessary, use the Binder program in the TNS environment or the native link editor utility (`nld`) in the TNS/R environment to combine multiple object files into one executable object file.
3. If you compiled the program in the TNS environment but plan to run it in the TNS/R environment, consider running the Accelerator for the C object file as an optional step to optimize the object code.
4. Run the SQL compiler (SQLCOMP) to compile the SQL source statements in the C object file and to validate the output SQL program file for execution.
5. Run the SQL program file from a terminal using the TACL RUN (or RUND) command or from a process using a system procedure such as NEWPROCESS or PROCESS\_CREATE\_.

Version 315 (or later) SQL/MP software supports the development of C 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](#).

## Processing Errors, Warnings, and Status Information

NonStop SQL/MP returns error and status information to a host-language program after the execution of each embedded SQL statement or directive. NonStop 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 run 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

With static SQL statements, you code the actual SQL statement in the C source file. However, with dynamic SQL, a C program can construct, compile, and run 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, and then run by an EXECUTE or EXECUTE IMMEDIATE statement.

[Example 1-2](#) shows a simple example of a dynamic INSERT statement (which is similar to the static SQL INSERT statement in [Example 1-1](#) on page 1-4). This program example dynamically builds an INSERT statement that inserts information into the PARTS table from information entered by a user.

---

## Example 1-2. Dynamic SQL Statements in a C Program

```
/* C source file */
...
char intext[201];
EXEC SQL BEGIN DECLARE SECTION;
    char operation[201];
EXEC SQL END DECLARE SECTION;

void dynamic_insert_function(void)
{
    ...
    /* User enters INSERT statement in the intext variable. */
    strncpy (operation, intext, 201);
    EXEC SQL EXECUTE IMMEDIATE :operation;
}
```

---

At run time, the program prompts a user for information to build the INSERT statement. The user enters this information in the INTEXT variable:

```
INSERT INTO $vol5.sales.parts (partnum, price, partdesc)
VALUES (4120, 60000.00, "V8 DISK OPTION")
```

The program moves the statement to the host variable OPERATION. The program has declared OPERATION as a host variable so that it is available to both SQL and C statements. The program then uses the EXECUTE IMMEDIATE statement to compile and run the INSERT statement in OPERATION. (This program could also have used the PREPARE and EXECUTE statements to compile and run the statement.)

For more information, see [Section 10, Dynamic SQL Operations](#).



# SQL/MP Version Management

Each product version update (PVU) of NonStop SQL/MP has an associated version number. The first two PVUs were version 1 (C10 and C20) and version 2 (C30). Version 300 SQL/MP began using a three-digit version number to allow for software product revisions (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 340 might be 345, 350, and 360.

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 version 2 and version 315 SQL/MP software, but it is not compatible with version 1 software.

This manual includes this version information:

- Using compatible versions of the C compiler, SQL compiler, and SQL executor to compile and run 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
- Using run-time SQLSA versioning, which allows a program to use an SQLSA structure with the same version as the current SQL/MP software for the system (available with version 340 or later SQL/MP software)
- Converting a C program written for version 1 or version 2 SQL/MP software to use version 300 (or later) SQL features and data structures

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



# 2 Host Variables

A host variable is a data item you can use in both C statements and NonStop SQL/MP statements to allow communication between the two types of statements. A host variable appears as a C identifier and can be any C data item declared in a Declare Section that has a corresponding SQL/MP data type as shown in [Table 2-1](#) on page 2-3 and [Table 2-2](#) on page 2-4. However, a host variable cannot be the name or identifier (the left part) of a `#define` directive.

For static SQL operations, a host variable can be an input or an output variable (or both in some cases) in an SQL statement. An input host variable transfers data from the program to the database, whereas an output host 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 a 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](#)
- [Coding Host Variable Names](#) on page 2-2
- [Using Corresponding SQL and C Data Types](#) on page 2-3
- [Specifying Host Variables in SQL Statements](#) on page 2-6
- [Declaring and Using Host Variables](#) on page 2-7
- [Using Indicator Variables for Null Values](#) on page 2-17
- [Creating Host Variables Using the INVOKE Directive](#) on page 2-18
- [Associating a Character Set With a Host Variable](#) on page 2-24

## 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.

- Place a Declare Section with the C variable declarations. 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 C structure declaration.
- Specify the C `#include` directive in a Declare Section to copy declarations from another file. However, do not use the SQL SOURCE directive.
- Use either C or SQL comment statements in a Declare Section.

## Coding Host Variable Names

Use C naming conventions for your host variable and indicator variable names. A name can contain from 1 to 31 alphanumeric characters, including the underscore (`_`), and must begin with a letter or an underscore. To avoid conflicts with HP names, do not begin your names with two underscores or end them with one underscore. This example uses a Declare Section with host variable names:

```
EXEC SQL BEGIN DECLARE SECTION;

short order_number;           /* simple variables */
char host_var_for_sql_statement;

struct employee               /* structure          */
{
    short empnum;
    char first_name[16];
    union {
        char last_name[21];
        char name_code_item[3];
        } union_last_name;
    short deptnum;
    short jobcode;
} employee_info;
int *ptr_to_table;           /* pointer */

#include copyfile             /* copy file */

EXEC SQL END DECLARE SECTION;
```

# Using Corresponding SQL and C Data Types

[Table 2-1](#) and [Table 2-2](#) on page 2-4 list the corresponding SQL and C data types.

**Table 2-1. Corresponding SQL and C Character Data Types**

SQL/MP Data Type	C Data Type
<b>Fixed-Length Character Data Type **</b>	
<i>hostvar</i> CHAR( <i>l</i> )	char <i>hostvar</i> [ <i>l</i> + 1]*
<i>hostvar</i> PIC X( <i>l</i> )	
<b>Fixed-Length Character Data Type With CHARACTER SET Clause</b>	
<i>hostvar</i> CHARACTER ( <i>l</i> ) CHARACTER SET <i>charset</i>	char CHARACTER SET <i>charset</i> <i>hostvar</i> [ <i>l</i> + 1]*
<i>hostvar</i> PIC X( <i>l</i> ) CHARACTER SET <i>charset</i>	
<b>Fixed-Length Character Data Type With NATIONAL CHARACTER Clause</b>	
<i>hostvar</i> NATIONAL CHARACTER ( <i>l</i> )	char CHARACTER SET <i>charset</i> <i>hostvar</i> [ <i>l</i> + 1]*
<b>Variable-Length Character Data Type **</b>	
<i>hostvar</i> VARCHAR( <i>l</i> )	struct { short <i>len</i> ; char <i>val</i> [ <i>l</i> + 1];* } <i>hostvar</i> ;
<b>Variable-Length Character Data Type With CHARACTER SET Clause</b>	
<i>hostvar</i> VARCHAR ( <i>l</i> ) CHARACTER SET <i>charset</i>	struct { short <i>len</i> ; char CHARACTER SET <i>charset</i> <i>val</i> [ <i>l</i> + 1];* } <i>hostvar</i> ;
<b>Variable-Length Character Data Type With NATIONAL CHARACTER Clause</b>	
<i>hostvar</i> NATIONAL CHARACTER VARYING( <i>l</i> )	struct { short <i>len</i> ; char CHARACTER SET <i>defcharset</i> <i>val</i> [ <i>l</i> + 1];* } <i>hostvar</i> ;
<i>hostvar</i>	A host variable name; <i>hostvar</i> must follow the naming conventions for a C identifier.
<i>l</i>	A positive integer that represents the length in characters of the host variable.
<i>len, val</i>	The length and value of the host variable.
<i>charset</i>	One of these character-set keywords: KANJI, KSC5601, ISO8859 <i>n</i> , where <i>n</i> is 1 – 9, or UNKNOWN (a single-byte unknown character set).
<i>defcharset</i>	The system default multibyte character set; <i>defcharset</i> is KANJI, unless it is otherwise set or changed during system generation.
*	An extra byte is generated as a place holder for a null terminator.
**	If a character set is not specified, the character set is UNKNOWN.

**Table 2-2. Corresponding SQL and C Numeric, Date-Time, and INTERVAL Data Types**

SQL/MP Data Type	C Data Type
<b>Numeric Data Types</b>	
NUMERIC (1 to 4, <i>s</i> ) SIGNED	short
NUMERIC (1 to 4, <i>s</i> ) UNSIGNED	unsigned short
NUMERIC (5 to 9, <i>s</i> ) SIGNED	long
NUMERIC (5 to 9, <i>s</i> ) UNSIGNED	unsigned long
NUMERIC (10 to 18, <i>s</i> ) SIGNED	long long
PIC 9( <i>l-s</i> )V9( <i>s</i> ) COMP	Same as NUMERIC
DECIMAL (1, <i>s</i> )	decimal[ <i>l</i> + 1]**
PIC 9( <i>l-s</i> )V9( <i>s</i> )	decimal[ <i>l</i> + 1]**
SMALLINT SIGNED	short
SMALLINT UNSIGNED	unsigned short
INTEGER SIGNED	long
INTEGER UNSIGNED	unsigned long
LARGEINT SIGNED	long long
FLOAT (1 to 22 bits)	float
REAL	float
FLOAT (23 to 54 bits)	double
DOUBLE PRECISION	double
<b>Date-Time and INTERVAL Data Types</b>	
DATETIME, TIMESTAMP, DATE, TIME	char[ <i>l</i> + 1]*
INTERVAL	char[ <i>l</i> + 1]***
<i>l</i>	A positive integer that represents the length. For DECIMAL, <i>l</i> must range from 1 – 18.
<i>s</i>	A positive integer that represents the scale of the number.
*	An extra byte is generated as a place holder for a null terminator.
**	The decimal data type is normally used to declare an array that can hold all the digit characters, the sign, and, optionally, a null terminator. The size of the array should be no more than 20 (19 plus an extra byte for the null terminator), or 21 (20 plus an extra byte for the null terminator) if a separate sign is used.
***	An INTERVAL data type has an extra byte for a sign.

---

**Note.** C programs that contain an embedded SQL/MP code do not support the use of unsigned `long long` C variables even if that data type is not used for the SQL query.

C programs containing unsigned `long long` C variables outside the EXEC SQL statements cannot be compiled in the Guardian and OSS environments. A workaround is to use the PC cross compiler. C programs with unsigned `long long` variables within the EXEC SQL statements cannot be compiled because NonStop SQL/MP does not support the unsigned `long long` data type.

---

## Data Conversion

NonStop SQL/MP performs the conversion between SQL and C data types:

- When a host variable serves as an input variable (supplies a value to the database), NonStop 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), NonStop SQL/MP converts the value to the data type of the host variable.

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

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

If an input value is too large for an SQL column, NonStop SQL/MP returns error 8300 (file system error encountered). If you are using the `SQLCADISPLAY` procedure to obtain an error message, `SQLCADISPLAY` also returns file-system error number 1031.

For numeric types, NonStop SQL/MP converts data between signed and unsigned types and between types with different precisions. Use the `SETSCALE` function to communicate a number's scale to and from a database.

---

**Note.** For optimal performance, declare host variables with corresponding data types and the same lengths as their respective columns in SQL statements (with consideration for the extra byte required for the null terminator). This programming practice minimizes the data conversion performed by NonStop 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 C compiler to handle the name as a host variable. To use a pointer as a host variable in SQL statements, place the colon before the asterisk.

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

*hostvar*

is the host variable name; *hostvar* can be any valid C identifier with a C data type that corresponds to an SQL data type, but it cannot be on the left-hand side of a #define directive. Precede *hostvar* with a colon (:) in an SQL statement.

INDICATOR

is a keyword that must precede *indicator\_hostvar*.

*indicator\_hostvar*

is an indicator variable of type short. Precede *indicator\_hostvar* with a colon (:) in an SQL statement.

For values returned to a host variable, *indicator\_hostvar* is -1 if the value is null or 0 if the value is not null. To insert null values into the database, set *indicator\_hostvar* to a value less than zero.



**TYPE AS**

specifies that the host variable will have the specified date-time (DATETIME, DATE, TIME, or TIMESTAMP) or INTERVAL data type. If a host variable must contain date-time or INTERVAL values, define it as a character data type. To cause NonStop SQL/MP to handle the host variable as a scaled value, either use the SETSCALE function or define the variable as C data type fixed.

## Declaring and Using Host Variables

You can declare and use these data types as host variables:

- Fixed and variable length character data types (CHAR and VARCHAR)
- Structures
- Decimal data types
- Fixed-point data types
- Date-time and INTERVAL data types

### Fixed-Length Character Data

The C language uses a character array plus a null terminator (`\0`) to store a string literal. Most C string-handling routines (for example, `strlen` and `printf`) require the null terminator. Follow these guidelines when you use character arrays as host variables for string literals.

### Declaring a Character Array

When you declare a character array as a host variable, the C compiler reserves the last byte of the array as a place holder for a null terminator. Therefore, declare a character array one byte longer than the actual number of characters. (The INVOKE directive automatically appends an extra byte to a character array, provided you do not specify the `CHAR_AS_ARRAY` option in the `SQL` pragma.) This declaration is for an SQL column up to 20 bytes long:

```
EXEC SQL BEGIN DECLARE SECTION;
    char last_name[21];          /* 20-byte last name */
EXEC SQL END DECLARE SECTION;
...
```

### Selecting Character Data

When selecting character data from a database to return to a host variable array, NonStop SQL/MP does not append a null terminator to the data. Therefore, before using the array in a C string-handling routine that requires a null terminator, you must append a null terminator to the array. This example selects character data from the

SHIPMENTS table and appends a null terminator to the `prod_desc` array before printing the data:

```
EXEC SQL BEGIN DECLARE SECTION;
    short    prod_num;
    char     prod_desc[11];
EXEC SQL END DECLARE SECTION;
...

EXEC SQL
    SELECT prod_num, prod_desc INTO :prod_num, :prod_desc
    FROM =shipments WHERE prod_num > min_num;
...
/* append null terminator before displaying string */
prod_desc[11] = "\0";
printf("%d %s\n", prod_num, prod_desc);
```

## Inserting Character Data

In an SQL/MP database, fixed-length character columns are always padded with blanks. Therefore, if the number of characters in an array is less than the size of the character column, pad the array with blanks before inserting it into the database. Otherwise, the INSERT statement stores the null terminator in the database, and comparison operations fail. This example inserts data into the PRODUCTS table. The `prod_desc` array is six bytes long (five bytes for the column value and one byte for the null terminator).

```
void function(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char prod_desc[6]; /* Use for a 5-character column */
    EXEC SQL END DECLARE SECTION;
    memcpy(prod_desc, "abc  ", 5); /* copy 5 characters */
                                /* (abc plus 2 blanks) */
    ...
    EXEC SQL INSERT INTO =products VALUES (:prod_desc);
}
```

This example pads the `prod_desc` array with blanks before it inserts the array into the database:

```
/* Routine to pad an array of characters */
/* with blanks on the right. */
void blank_pad(char *buf, size_t size)
{
    size_t i;
    i = strlen(buf);
    if (i < size)
        memset(&buf[i], ' ', size - i);
}

void function(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
```

```

char prod_desc[6];          /* use for 5-character column */
EXEC SQL END DECLARE SECTION;

strcpy(prod_desc, "abc"); /* Copy 3 characters and      */
                        /* null terminator      */

...
/* Do not include space for null byte in the size      */
blank_pad(prod_desc, sizeof prod_desc - 1);
EXEC SQL INSERT INTO =products VALUES (:prod_desc);
}

```

## Variable-Length Character Data

The VARCHAR data type represents one data element; however, the C compiler converts the type to a structure with two data items. The C compiler derives the group item name from the VARCHAR column name and the names of the subordinate data items, where:

- `len` is a numeric data item that represents the length.
- `val` is a fixed-length character data item for the string, plus an extra byte for the null terminator, if the SQL pragma specifies the `CHAR_AS_STRING` option.

For example, if a column `CUSTNAME` is defined as `VARCHAR(26)`, and the SQL pragma specifies the `CHAR_AS_STRING` option, INVOKE generates this structure:

```

struct
{
    short len;
    char  val[27];
} custname;

```

You can refer to the individual data items or the structure name as host variables.

If you explicitly declare a structure as a host variable for a VARCHAR column (rather than using INVOKE), declare the length as a `short` data type (and not an `int`).

## Structures

You can refer to a structure name as a host variable only if the structure corresponds to a VARCHAR data type. For structures that do not correspond to a VARCHAR data type, the fields within the structure are the host variables. However, when you refer to an individual field name in the structure, you must include the structure name with the field name. For example, the structure `employee_info` contains the `empid` and `empname` fields:

```

EXEC SQL BEGIN DECLARE SECTION;
struct employee
{
    long  empid;
    char  empname[21];
} employee_info;
EXEC SQL END DECLARE SECTION;

```

To use a field as a host variable in an SQL statement, refer to the field by using the structure name:

```
EXEC SQL SELECT empid, empname  
        INTO :employee_info.empid, :employee_info.empname  
        ... ;
```

## Decimal Data Types

Use the DECIMAL data type for ASCII numeric data. Because a decimal string is actually a fixed-length character string that contains only ASCII digits, considerations for fixed-length character strings also apply to decimal strings. Follow these guidelines when you use character arrays as host variables for DECIMAL data:

- Declare a decimal array one byte larger than the number of digits you expect to store in the array.
- Append a null terminator to an SQL/MP decimal string before you process it as a C decimal string.
- Right justify a C decimal string and pad the string on the left with ASCII zeros up to the length of the corresponding SQL column before you insert the value into the database.

HP C does not support direct manipulation of decimal strings. To perform C arithmetic operations on SQL columns of DECIMAL data type, first convert the column to an integral type using the `dec_to_longlong` routine.

HP C also provides the `longlong_to_dec` routine to convert type long long to type decimal. Although the `longlong_to_dec` routine supports a variety of formats for signed decimal strings, NonStop SQL/MP supports only the embedded leading signed format. Therefore, always specify the embedded leading signed format when you intend to pass the converted decimal string to NonStop SQL/MP.

For more information about C routines, see the *C/C++ Programmer's Guide*.

## Fixed-Point Data Types

HP C does not have a data type that maps directly to a fixed-point number (that is, an SQL numeric data type with scale). If you transfer fixed-point values to integral or floating-point host variables, consider these guidelines:

- When you transfer a fixed-point value to a host variable of floating-point data type, NonStop SQL/MP converts the fixed-point value to a floating-point value and generates a warning to indicate a loss of precision.
- When you transfer a fixed-point value into an integer host variable, NonStop SQL/MP stores the integral part of the value and generates a warning to indicate a loss of data (the fractional part). To retain the fractional part, use the SETSCALE function to scale the fixed-point value before transferring it to the host variable.

## SETSCALE Function

The SETSCALE function directs NonStop SQL/MP to use a host variable in SQL statements as if the host variable were declared with a specific scale. Use the SETSCALE function for these operations:

- To insert scaled values (for example, prices) into a database

- To select database values into host variables
- To refer to values stored in the database for comparisons

The SETSCALE function has this syntax:

```
SETSCALE ( :host-variable
          [ [ INDICATOR ] :indicator-variable ] , scale )
```

*host-variable*

is an integer host variable.

*indicator-variable*

is an indicator variable associated with the host variable.

*scale*

specifies the scale of *host-variable*. The values for *scale* depend on the size of *host-variable*:

Size	Values
2-byte integers	0 – 5 decimal digits
4-byte integers	0 – 10 decimal digits
8-byte integers	0 – 18 decimal digits

Follow these guidelines when you use the SETSCALE function:

- If you are transferring a value from a host variable to a database using an INSERT or UPDATE statement, you must assign a value to the host variable that allows for the scale. For example, to insert a price of \$123.45, assign 12345 to hostvar and specify a scale of 2.
- If you are retrieving a value from a database using a SELECT statement, NonStop SQL/MP returns a value that allows for the scale in the host variable. For example, if your program specifies a scale of 2 in the SELECT statement and 123.45 is stored in the database, SQL/MP returns 12345 to the host variable.
- The scale is valid only for SQL statements. If you use the SETSCALE function in SQL statements and the host variables in calculations using C statements, the C statements must handle the scale.
- To use SETSCALE in an expression, apply the SETSCALE function to each operand individually rather than to the result of the expression. For example, this expression adds two prices with a scale of 2 decimal places:

```
SETSCALE (:price1, 2) + SETSCALE (:price2, 2)
```

- When you use the INVOKE directive for a column with a scaled data type, the C compiler generates a comment that shows the scale of the column. For example, for price with data type NUMERIC (8,2), INVOKE generates the following:

```
long      price; /* scale is 2 */
```

These examples use the `=parts` DEFINE to represent the PARTS table. The first example inserts a new row with the value 98.34 in the PARTS.PRICE column after storing the value in the host variable `host_var1`. The value is multiplied by 100 for storing as a whole number.

```
host_var1 = 9835;
EXEC SQL INSERT INTO =parts (price)
        VALUES ( SETSCALE (:host_var1, 2) ) ;
```

The next example updates the PARTS.PRICE column for a disk controller to 158.34. The value is multiplied by 100 and stored in the host variable `host_var2`.

```
host_var2 = 15834;
EXEC SQL UPDATE =parts
        SET price = SETSCALE (:host_var2, 2)
        WHERE parts.partdesc = "disk controller" ;
```

The next example retrieves the value in the PARTS.PRICE column for a disk controller and stores the value in the host variable `host_var3`. The value has a scale of 2.

```
EXEC SQL SELECT parts.price INTO SETSCALE ( :host_var3, 2 )
        FROM =parts
        WHERE parts.partdesc = "disk controller" ;
```

The next example retrieves the part description for the part with a price of 999.50. The PARTS.PRICE value is stored in the host variable `host_var4` and passed to NonStop SQL/MP in the search condition. The retrieved value is stored in the host variable `host_varstore`.

```
host_var4 = 99950;
EXEC SQL SELECT parts.partdesc INTO :host_varstore
        FROM =parts
        WHERE parts.price = SETSCALE (:host_var4,2);
```

## Date-Time and INTERVAL Data Types

[Table 2-3](#) describes the SQL/MP date-time and INTERVAL data types you can use for host variables.

---

**Table 2-3. Date-Time and INTERVAL Data Types** (page 1 of 2)

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 a synonym for DATETIME YEAR TO DAY

---

**Table 2-3. Date-Time and INTERVAL Data Types** (page 2 of 2)

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

To communicate date-time or INTERVAL values between C and SQL statements, declare a character array as a host variable and then use the TYPE AS clause to cause NonStop SQL/MP to interpret the value as a date-time or INTERVAL value. For the syntax of the TYPE AS clause, see [Specifying Host Variables in SQL Statements](#) on page 2-6.

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 08/15/1996, 1996-08-15, or 15.08.1996. You control the display format by inserting the value in the format you want and retrieving the value using the DATEFORMAT function. You must declare the host variable size to be consistent with the format you plan to use.

This example inserts date-time values into the BILLINGS table:

```
EXEC SQL BEGIN DECLARE SECTION;
    struct billing_rec
    {
        char custnum[4];
        char start_date[11];
        char billing_date[11];
        char time_before_pmt[5];
    };
    struct billing_rec billings = { ' ',' ',' ',' ',' ',' ' };
...
EXEC SQL END DECLARE SECTION;
...
strcpy(billings.billing_date, "1996-08-20");
strcpy(billings.time_before_pmt, " 90");
...
EXEC SQL
    INSERT INTO billings VALUES
        ("923", DATE "1985-10-15",
         :billing_date TYPE AS DATE,
         :time_before_pmt TYPE AS INTERVAL DAY);
...
```

When you invoke a column with a date-time (DATETIME, DATE, TIME, or TIMESTAMP) or INTERVAL data type, the data is represented as a character field. The size of the field is determined by the range of the date-time or INTERVAL column. You control the display format by inserting the value in the format you want and retrieving the value using the DATEFORMAT function. If you use INVOKE to generate host variables from an SQL table definition, you can specify the DATEFORMAT clause to determine the size.



INTERVAL values are represented as character strings with a separator between the values of the fields (year-month or day-time). An extra byte is generated at the beginning of the INTERVAL string for a sign. The default representations for DATE and INTERVAL values are shown in these examples.

## DATE Representation

The column definition and representation in the table for December 22, 1988 is:

birth\_date DATE

Year				Separator	Month		Separator	Day		Null
1	9	8	8	-	1	2	-	2	2	

012

If the DATEFORMAT clause on the INVOKE directive specifies DEFAULT, a column with the range of fields YEAR TO DAY is represented as an 11-character string (10 characters plus a byte for a null character). The C compiler creates this structure:

```
struct employee_rec {
    char name[18];
    char birth_date[11];
};
```

## INTERVAL Representation

The column definition and representation in the table for 36 years, 7 months is:

AGE INTERVAL YEAR(2) TO MONTH

Sign	Year		Separator	Month		Null
+	3	6	-	0	7	

013

The C compiler creates this structure:

```
struct employee_rec {
    char name[21];
    char age[7];
};
```

## Example—Creating DATETIME and INTERVAL Data Types

---

### Example 2-1. Creating Valid DATETIME and INTERVAL Data Types

```
#include <stdio.h>
#include <string.h>
#include <sql.h>

#define STMT_LEN 256

EXEC SQL BEGIN DECLARE SECTION;
short sqlcode;
char hv_projdesc[30];
char hv_start_date[11];
char in_start_date[11];
char curspec[STMT_LEN];
EXEC SQL END DECLARE SECTION;

int main()
{
    int len;
    strcpy(curspec,
           "SELECT projdesc, CAST(start_date AS CHAR(10)) FROM test1 "
           "WHERE start_date <= CAST(CAST( ? AS CHAR(10)) "
           "AS DATE) BROWSE ACCESS");
    len = strlen(curspec);
    memset(&curspec[len], ' ', STMT_LEN - len);

    EXEC SQL PREPARE cursor_spec from :curspec;

    /* Declare the dynamic cursor from the prepared statement. */
    EXEC SQL DECLARE get_proj CURSOR FOR cursor_spec;

    /* Initialize the parameter in the WHERE clause. */
    printf("Enter the most recent start date in the form yyyy-mm-dd: ");
    scanf("%s", in_start_date);

    /* Open the cursor using the value of the dynamic parameter. */
    EXEC SQL OPEN get_proj USING :in_start_date;

    /* Fetch the first row of the result table. */
    EXEC SQL FETCH get_proj INTO :hv_projdesc,:hv_start_date;

    while (sqlcode == 0)
    {
        hv_start_date[10]='\0';
        printf("\n Start Date: %s", hv_start_date);

        /* Fetch the next row of the result table. */
        EXEC SQL FETCH get_proj INTO :hv_projdesc,:hv_start_date;
    }
    /* Close the cursor. */
    EXEC SQL CLOSE get_proj;

    return 0;
}
```

---

# Using Indicator Variables for Null Values

A null value in an SQL column indicates that the value is either unknown for the row or is not applicable to the row. A program inserts a null value or tests for a null value using an indicator variable. An indicator variable is a 2-byte integer variable 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.

A program can use an indicator variable associated with a host variable:

- To insert 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 values into a database with an INSERT or UPDATE statement, a program sets the indicator variable to less than zero (0) for a null value or zero (0) for a nonnull value before executing the statement. This statement inserts values into the ODETAIL table. The columns UNIT\_PRICE and QTY\_ORDERED allow null values.

```
EXEC SQL INSERT INTO =odetail
    (ordernum, partnum, unit_price, qty_ordered)
VALUES ( :odetail.ordernum,
        :odetail.partnum,
        :odetail.unit_price      :odetail.unit_price_i,
        :odetail.qty_ordered     :odetail.qty_ordered_i );
```

## 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 values from the ODETAIL table and returns the values to host variables. After the SELECT statement runs, 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 SELECT ordernum, partnum, unit_price, qty_ordered
    INTO :odetail.ordernum,
        :odetail.partnum,
        :odetail.unit_price INDICATOR :odetail.unit_price_i,
        :odetail.qty_ordered INDICATOR
        :odetail.qty_ordered_i,
FROM sales.odetail
    WHERE ordernum = 300380 AND partnum = 2402 ;

...
if ((odetail.unit_price_i < 0)      ||
    (odetail.qty_ordered_i < 0))
    handle_null_value();
```

```
else display_result();
...
```

## Retrieving Rows With Null Values

You can 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. In this case, NonStop 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;
...
EXEC SQL OPEN get_null_salary ;
...

EXEC SQL FETCH get_null_salary INTO
    :employee_record.empnum,
    :employee_record.first_name,
    :employee_record.last_name,
    :employee_record.deptnum,
    :employee_record.jobcode,
    :employee_record.salary ;

/* Test SQLCODE. */
/* Process the row that contains the null salary. */
/* Branch back to FETCH the next row. */
...
EXEC SQL CLOSE get_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. INVOKE converts the column names to C identifiers and generates a C declaration for each column. When a column allows null values, INVOKE also creates an indicator variable for the column. For views only, INVOKE includes the system-defined primary keys in the definition. You can use a class MAP DEFINE name for a table or view name in an INVOKE directive, but not for a record name.

To run an INVOKE directive, a process started by the program must have read access to the invoked tables or views during C compilation. For details, see [Required Access Authority](#) on page 7-1.

The `CHAR_AS_STRING` and `CHAR_AS_ARRAY` options of the SQL pragma affect the INVOKE directive as follows:

- The `CHAR_AS_STRING` option (the default) causes INVOKE to generate character data types with an extra byte for a null terminator.
- The `CHAR_AS_ARRAY` option causes INVOKE to generate character data types without the extra byte for a null terminator.

## Advantages of Using an INVOKE Directive

You can declare a host variable as a C structure corresponding to an SQL table or view without using an INVOKE directive. However, using an INVOKE directive to generate host variables has these advantages:

- Program independence—If you modify a table or view, the INVOKE directive re-creates 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 structure tag).
- Program performance—The INVOKE directive maps SQL data types to the corresponding C 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.

## C Structures Generated by the INVOKE Directive

These examples show the correspondence between tables `TYPESC1` and `TYPESC2` that contain columns of various SQL data types and the C structures generated by the INVOKE directive. [Example 2-2](#) on page 2-20 shows the CREATE TABLE statements that generate the tables, and [Example 2-3](#) on page 2-21 shows the structures generated by the INVOKE directives.

**Example 2-2. CREATE TABLE Statements**


---

```

CREATE TABLE \NEWYORK.$DISK1.SQL.TYPESC1 (
type_char          CHAR          (10)          NOT NULL,
type_char_null     CHAR          (10)
,
type_varchar       VARCHAR      (10)          NOT NULL,
type_varchar_null  VARCHAR      (10)
,
type_num4_s        NUMERIC      (4)          SIGNED          NOT NULL,
type_num4_u        NUMERIC      (4)          UNSIGNED         NOT NULL,
type_num9_s        NUMERIC      (9,2)        SIGNED          NOT NULL,
type_num9_u        NUMERIC      (9,2)        UNSIGNED         NOT NULL,
type_num18_s       NUMERIC      (18,2)       SIGNED          NOT NULL,
type_small_s       SMALLINT          SIGNED          NOT NULL,
type_small_u       SMALLINT          UNSIGNED         NOT NULL,
type_int_s         INTEGER          SIGNED          NOT NULL,
type_int_u         INTEGER          UNSIGNED         NOT NULL,
type_large_s       LARGEINT         SIGNED          NOT NULL,
type_dec_s         DECIMAL      (18,2)       SIGNED          NOT NULL,
type_dec_u         DECIMAL      (9,2)       UNSIGNED         NOT NULL,
type_pic_s         PIC 9(9)          COMP          NOT NULL,
type_picx          PIC X(10)         NOT NULL,
type_picx_long     PIC XXXXXXXXXXXXXXXXXXXXX NOT NULL,
type_float_15      FLOAT (15)        NOT NULL,
type_float_30      FLOAT (30)        NOT NULL,
type_real          REAL              NOT NULL,
type_dbl_prec      DOUBLE PRECISION  NOT NULL,
type_datetime      DATETIME YEAR TO DAY NOT NULL,
type_date          DATE              NOT NULL,
type_time          TIME              NOT NULL,
type_timestamp     TIMESTAMP         NOT NULL,
type_interval      INTERVAL YEAR TO MONTH NOT NULL,
type_char_null_ok  CHAR(10)          DEFAULT NULL,
type_num_null_ok   SMALLINT          DEFAULT NULL
) CATALOG $SQL.SQLCAT ;
CREATE TABLE \NEWYORK.$DISK1.SQL.TYPESC2 (
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_nchar         NCHAR          (10)          NOT NULL,
type_nchar_v       NCHAR VARYING (10)          NOT NULL
type_varchar1      VARCHAR      (10) CHARACTER SET ISO88591 NOT
NULL,
type_varchar2      VARCHAR      (10) CHARACTER SET KANJI   NOT NULL,
type_picx1         PIC X(10) CHARACTER SET ISO88591 NOT NULL,
type_picx2         PIC X(10) CHARACTER SET KANJI   NOT NULL
) CATALOG $SQL.SQLCAT ;

```

---

These INVOKE directives are coded in a C source file:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE \newyork.$disk1.sql.typescl AS typescl_struct;
EXEC SQL INVOKE \newyork.$disk2.sql.typescl AS typescl2_struct;
EXEC SQL END DECLARE SECTION;
```

---

**Example 2-3. Structures Generated by the INVOKE Directive** (page 1 of 2)

```
/* Record Definition for table \NEWYORK.$DISK1.SQL.TYPESCL
*/
/* Definition current at 13:52:15 - 8/27/96 */
struct typescl_type {
    char            type_char[11];
    short           type_char_null_i;
    char            type_char_null[11];
    struct {
        short       len;
        char        val[11];
    } type_varchar;
    short           type_varchar_null_i;
    struct {
        short       len;
        char        val[11];
    } type_varchar_null;
    short           type_num4_s;
    unsigned short  type_num4_u;
    long            type_num9_s;           /* scale is 2 */
    unsigned long   type_num9_u;          /* scale is 2 */
    long long       type_num18_s;         /* scale is 2 */
    short           type_small_s;
    unsigned short  type_small_u;
    long            type_int_s;
    unsigned long   type_int_u;
    long long       type_large_s;
    decimal         type_decs[19];        /* scale is 2 */
    decimal         type_dec_u[10];       /* scale is 2 */
    unsigned long   type_pic_s;
    char            type_picx[11];
    char            type_picx_long[21];
    float           type_float_15;
    double          type_float_30;
    float           type_real;
    double          type_dbl_prec;
    char            type_datetime[11];
    char            type_date[11];
```

---

---

**Example 2-3. Structures Generated by the INVOKE Directive** (page 2 of 2)

---

```

char          type_time[9];
char          type_timestamp[27];
char          type_interval[7];
short        type_char_null_ok_i;
char          type_char_null_ok[11];
short        type_num_null_ok_i;
short        type_num_null_ok; };
/* Record Definition for table \NEWYORK.$DISK1.SQL.TYPESC2
*/
/* Definition current at 13:52:19 - 8/27/96 */
struct typesc2_type {
char          CHARACTER SET ISO88591 type_char1[11];
short        type_char1_null_i;
char          CHARACTER SET ISO88591
type_char1_null[11];
char          CHARACTER SET KANJI type_char2[11];
short        type_char2_null_i;
char          CHARACTER SET KANJI type_char2_null[11];
char          CHARACTER SET KANJI type_nchar[11];
struct {
short        len;
char          CHARACTER SET KANJI val[11];
} type_nchar_v;
struct {
short        len;
char          CHARACTER SET ISO88591 val[11];
} type_varchar1;
struct {
short        len;
char          CHARACTER SET KANJI val[11];
} type_varchar2;
char          CHARACTER SET ISO88591 type_picx1[11];
char          CHARACTER SET KANJI type_picx2[11];
};

```

---

## Using Indicator Variables With the INVOKE Directive

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

If a column name is 30 or 31 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 cause INVOKE to generate an indicator variable name derived from the column name and the prefix or suffix. This example shows an INVOKE directive with the PREFIX and SUFFIX clauses as it appears in a C source program:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE ctable PREFIX beg_ SUFFIX _end;
EXEC SQL END DECLARE SECTION;
```

The C compiler generates this structure:

```
/* Record Definition for table \SYS.$DSK.PERSNL.CTABLE */
/* Definition current at 15:32:39 - 09/22/95 */
struct ctable_type {
    short      beg_znum_end;
    long       znum;
    short      beg_zchar_end;
    char       zchar[16];
};
```

## NULL STRUCTURE Clause

The NULL STRUCTURE clause causes INVOKE to generate a structure for a column that contains an indicator variable. The NULL STRUCTURE clause assigns the name indicator to all indicator variables in the structure.

This example shows an INVOKE directive with the NULL STRUCTURE clause as it appears in a C source program:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE emptbl AS emptbl_rec NULL STRUCTURE;
...
EXEC SQL END DECLARE SECTION;
...
```

The C compiler generates this structure:

```
/* Record Definition for table \SYS.$VOL.SUBVOL.EMPTBL */
/* Definition current at 16:07:00 - 05/17/94 */

struct emptbl_rec {
    unsigned short empnum;
    struct {
        short      indicator;
        char       valu[16];
    } first_name;
    char          last_name[21];
    struct {
        short      indicator;
        char       valu[11];
    } retire_date;
};
```

## Using INVOKE With SQLCI

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

```
>> INVOKE =dept FORMAT C TO copylib (deptrec);
...
```

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, when necessary, you can edit the host variables before copying them into your program's compilation unit.

Use the #include directive, not the SQL SOURCE directive to copy the host variable declarations in your program's compilation unit.

## Associating a Character Set With a Host Variable

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

```
char [ CHARACTER SET [ IS ] charset ] hostvar [ length ]
```

**CHARACTER SET [ IS ]**

are keywords that must precede the character set name. You must specify the CHARACTER SET clause in uppercase letters. If you omit the clause, the character set defaults to UNKNOWN.

*charset*

is the character set name, which must be one of these keywords (in uppercase letters): ISO8859<sub>*n*</sub> (*n* ranges from 1 through 9), KANJI, KSC5601, or UNKNOWN. The UNKNOWN keyword indicates an unknown single-byte character set and is equivalent to omitting the CHARACTER SET clause.

*hostvar*

is the name of the host variable, which must follow the naming conventions for a C identifier.

*length*

is the length in characters (not bytes) of the host variable. *length* must also include an extra byte for the null terminator, if the SQL pragma specifies the CHAR\_AS\_STRING option (the default).

---

**Note.** NonStop SQL/MP does not support the C wchar\_t data type.

---

## Treatment in C Statements

A C statement treats a host variable declared with the CHARACTER SET clause as if the host variable had been declared without the clause. A C statement also treats the host variable length as the specified length multiplied by the number of bytes per character plus the null terminator if the SQL pragma specifies the CHAR\_AS\_STRING option (the default). These examples show this treatment for single-byte and double-byte character set declarations:

### Host Variable Declaration

```
char CHARACTER SET ISO88591 hostv1[5]
char CHARACTER SET KANJI hostv2[10]
```

### Treatment in C Statements

```
char hostv1[5]
char hostv2[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` in the next example) of the VARCHAR group item to the host variable length in bytes and not characters.

For example, this host variable declaration specifies the double-byte KANJI character set for `emp_name`. The C assignment statement sets the length (`emp_name.len`) of the host variable name to 16 characters because the name (`emp_name.val`) contains 8 double-byte characters (which are represented as "c1c2c3c4c5c6c7c8").

```
EXEC SQL BEGIN DECLARE SECTION;
struct {
    short len;
    char CHARACTER SET KANJI val[10];
} emp_name;
EXEC SQL END DECLARE SECTION;

...
/* Insert data into the data base. */

strcpy (emp_name.val, "c1c2c3c4c5c6c7c8");
emp_name.len = strlen(emp_name.val);
EXEC SQL
    INSERT INTO =employee VALUES (:emp_name);
...

/* Select data from the data base. */

EXEC SQL
    SELECT employee_name INTO :emp_name FROM =employee;
emp_name.val[emp_name.len] = '\0';
...
```

The last C assignment statement sets the null terminator after the SELECT statement returns the employee name from the EMPLOYEE table.



# SQL/MP Statements and Directives

For a detailed description, including the syntax, of all SQL/MP statements and directives, see the *SQL/MP Reference Manual*.

This section includes:

- [Embedding SQL Statements](#)
- [Finding Information](#) on page 3-3

## Embedding SQL Statements

Use this syntax to embed a NonStop SQL/MP statement or directive in a C source file.

```
EXEC SQL sql-statement-or-directive ;
```

*sql-statement-or-directive*

is any SQL statement or directive shown in [Table 3-1](#) on page 3-3. The statement or directive must begin with the keywords EXEC SQL and end with a semicolon (;).

## Coding Statements and Directives

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

- Do not nest SQL statements or directives.
- Use only SQL comments in SQL statements and directives. SQL comments begin with a double hyphen (--) and end with the end of the line. You cannot use C comments in SQL statements or directives.
- Use only the C string delimiter, a double quote ("), for quoted strings.
- Code an SQL statement or directive on a single source code line or over several lines:

```
EXEC SQL WHENEVER SQLERROR :handle_error;
```

```
EXEC SQL DROP TABLE \ny.$disk1.invent.supplier;
```

```
EXEC SQL
  SELECT customer.custname
  INTO :customer.custname
```

```

        FROM =customer
        WHERE custnum = :find_this_customer

;

```

## Placing Statements and Directives

Place SQL statements and directives and C compiler pragmas in a C source file.

### SQL Pragma

To use embedded SQL statements and directives in a C program, you must specify the SQL pragma before any SQL or C statements (except comment statements). You can specify the SQL pragma either in your source file or as a compiler option in the implicit TACL RUN command that starts the C compiler. This example uses the SQL pragma in a source code file:

```
#pragma SQL
```

This example uses the SQL pragma as a compiler option:

```
C / IN csrc, OUT $s.#clst, NOWAIT / cobj; SQL
```

After the SQL pragma, place other SQL statements and directives in a C source file as described in these paragraphs.

### C Variable Declarations

You can use these statements and directives with C variable declarations:

- BEGIN DECLARE SECTION and END DECLARE SECTION directives
- DECLARE CURSOR statements for static SQL operations
- INVOKE directive
- INCLUDE STRUCTURES directive
- INCLUDE SQLCA, INCLUDE SQLSA, and INCLUDE SQLDA directives

### C Executable Statements

You can use these statements with C executable statements:

- Data manipulation language (DML) statements
- Data control language (DCL) statements
- Data definition language (DDL) statements
- Data status language (DSL) statements
- Transaction control statements
- Dynamic SQL statements (including DECLARE CURSOR)

## Anywhere in the Program

You can use these directives anywhere in a C program:

- WHENEVER directives
- SQL SOURCE directive
- CONTROL directives

## Finding Information

[Table 3-1](#) lists SQL/MP statements and directives you can embed in a C program and indicates where each statement or directive is documented.

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

Statement or Directive	Manual *	Description
<b>Data Declaration Directives</b>		
BEGIN DECLARE SECTION	SQLRM SQLPM/C	Designates the beginning of host variable declarations.
END DECLARE SECTION	SQLRM SQLPM/C	Designates the end of host variable declarations.
INCLUDE STRUCTURES	SQLRM SQLPM/C	Specifies the version of SQL structures generated.
INCLUDE SQLCA	SQLRM SQLPM/C	Generates the SQLCA structure for run-time status and error information.
INCLUDE SQLDA	SQLRM SQLPM/C	Generates the SQLDA structure to receive information about input and output variables for dynamic SQL statements.
INCLUDE SQLSA	SQLRM SQLPM/C	Generates the SQLSA structure to receive execution statistics about DML or PREPARE statements.
INVOKE	SQLRM SQLPM/C	Generates a structure description of a table or view.
*This statement is documented in one or more of these manuals:		
SQLRM	SQL/MP Reference Manual	
SQLPM/C	SQL/MP Programming Manual for C	

**Table 3-1. Summary of SQL/MP Statements and Directives** (page 2 of 4)

<b>Statement or Directive</b>	<b>Manual *</b>	<b>Description</b>
<b>Data Definition Language (DDL) Statements</b>		
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.
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.
<b>Error Checking Directives</b>		
WHENEVER	SQLRM SQLPM/C	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 <i>SQL/MP Reference Manual</i>		
SQLPM/C <i>SQL/MP Programming Manual for C</i>		



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

<b>Statement or Directive</b>	<b>Manual *</b>	<b>Description</b>
<b>Data Manipulation Language (DML) Statements</b>		
CLOSE	SQLRM SQLPM/C	Terminates a cursor.
DECLARE CURSOR	SQLRM SQLPM/C	Defines a cursor.
DELETE	SQLRM SQLPM/C	Deletes rows from a table or view.
FETCH	SQLRM SQLPM/C	Retrieves a row from a cursor.
INSERT	SQLRM SQLPM/C	Inserts rows into a table or view.
OPEN	SQLRM SQLPM/C	Opens a cursor.
SELECT	SQLRM SQLPM/C	Retrieves data from tables and views.
UPDATE	SQLRM SQLPM/C	Updates values in columns of a table or view.
<b>Data Control Language (DCL) Statements</b>		
CONTROL EXECUTOR	SQLRM SQLPM/C	Specifies whether to process data using a single executor or multiple executors working in parallel.
CONTROL QUERY	SQLRM SQLPM/C	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 SQLPM/C	Specifies parameters that control locks, opens, buffers, access paths, join methods, and join sequences on tables and views.
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.
*This statement is documented in one or more of these manuals: SQLRM <i>SQL/MP Reference Manual</i> SQLPM/C <i>SQL/MP Programming Manual for C</i>		

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

<b>Statement or Directive</b>	<b>Manual *</b>	<b>Description</b>
<b>Data Status Language (DSL) Statements</b>		
GET CATALOG OF SYSTEM	SQLRM	Returns the name of a local or remote system catalog.
GET VERSION	SQLRM SQLPM/C	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 SQLPM/C	Returns the program catalog version (PCV), program format version (PFV), or host object SQL version (HOSV) of an SQL program file.
<b>Transaction Control Statements</b>		
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.
<b>Dynamic SQL Statements</b>		
DESCRIBE	SQLRM SQLPM/C	Returns information about output variables for a prepared statement.
DESCRIBE INPUT	SQLRM SQLPM/C	Returns information about input variables for a prepared statement.
EXECUTE	SQLRM SQLPM/C	Runs a prepared SQL statement.
EXECUTE IMMEDIATE	SQLRM SQLPM/C	Runs an SQL statement contained in a host variable.
PREPARE	SQLRM SQLPM/C	Compiles a DDL, DML, DCL, or DSL statement.
RELEASE	SQLRM	Deallocates memory for a dynamic SQL statement referred to through a host variable.
*This statement is documented in one or more of these manuals:		
SQLRM	SQL/MP Reference Manual	
SQLPM/C	SQL/MP Programming Manual for C	

[Table 3-2](#) summarizes the C compiler pragmas that apply to a C program containing embedded SQL statements and directives. For a description of all C compiler pragmas, see the *C/C++ Programmer's Guide*.

**Table 3-2. C Compiler Pragmas for SQL/MP**

Pragma	Manual*	Description
SQL	SQLPM/C CPG	<p>Indicates to the C compiler that a program contains embedded SQL statements and directives.</p> <p>Also specifies options for processing the SQL statements or directives:</p> <ul style="list-style-type: none"><li>● SQLMAP generates an SQL map in the listing.</li><li>● WHENEVERLIST writes active WHENEVER options to the listing file after each SQL statement is processed.</li><li>● 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.</li></ul>
SQLMEM	SQLPM/C CPG	<p>Controls the placement of SQL internal structures in either the user data segment or extended data segment.</p> <p>SQLMEM applies only to the C compiler on TNS systems. The native mode C (NMC) compiler on TNS/R systems ignores this pragma.</p>

---

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

CPG	<i>C/C++ Programmer's Guide</i>
SQLPM/C	<i>SQL/MP Programming Manual for C</i>



You can access data in an SQL/MP database using this Data Manipulation Language (DML) statements in a C program:

- Simple data manipulations—SELECT (single-row), INSERT, UPDATE, and DELETE statements
- Cursor operations—DECLARE CURSOR, OPEN, FETCH, and CLOSE statements where the cursor contains a SELECT, UPDATE, or DELETE statement

Topics include:

- [Opening and Closing Tables and Views](#) on page 4-2
- [Single-Row SELECT Statement](#) on page 4-4
- [Multirow SELECT Statement](#) on page 4-6
- [INSERT Statement](#) on page 4-8
- [UPDATE Statement](#) on page 4-10
- [DELETE Statement](#) on page 4-12
- [Using SQL Cursors](#) on page 4-14

[Table 4-1](#) provides some guidelines for using these statements.

---

**Table 4-1. SQL/MP Statements for Data Retrieval and Modification** (page 1 of 2)

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.

---

**Table 4-1. SQL/MP Statements for Data Retrieval and Modification** (page 2 of 2)

SQL/MP Statement	Description
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 the OPEN, FETCH, and CLOSE statements, which increases the number of 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

NonStop 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. NonStop SQL/MP opens a table or view when a host-language program runs 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 NonStop SQL/MP to close a table using the CLOSE TABLES option of the FREE RESOURCES statement.

By default, NonStop SQL/MP opens partitions of base tables and indexes only if they are needed by a program. To cause NonStop SQL/MP to open all indexes and partitions the first time a 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 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 runs 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, BUFFERED, LOCKLENGTH, MAXEXTENTS, SERIALWRITES, TABLECODE, or VERIFIEDWRITES
  - ALTER INDEX with ADD PARTITION, DROP PARTITION, or RENAME
  - ALTER INDEX with BUFFERED, MAXEXTENTS, TABLECODE, SERIALWRITES, or VERIFIEDWRITES
  - 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 up again can also cause the system to terminate a program's open.)
3. The program tries to run another SQL statement for the table or view.
4. The SQL executor tries to recover, as described next. 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 runs 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 next.

### 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 abnormally terminate 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 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 run.

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 abnormally terminate 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 using a cursor, see [Using SQL Cursors](#) on page 4-14.

To run a SELECT statement, a process started by the program must have read access to all tables, protection views, and the underlying tables of any shorthand views used in the statement. For details, see [Required Access Authority](#) on page 7-1.

Do not use an asterisk (\*) in a SELECT statement in a C 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.



NonStop SQL/MP returns these values to `sqlcode` after a `SELECT` statement:

<b>sqlcode Value</b>	<b>Description</b>
0	The <code>SELECT</code> statement was successful.
100	No rows qualified for the <code>SELECT</code> statement specification.
<0	An error occurred; <code>sqlcode</code> contains the error number.
>0 (!100)	A warning occurred; <code>sqlcode</code> 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 the unique value of a column (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 `find_this_customer`. (This example sets `find_this_customer` to customer number 5635 using an assignment statement, but in a typical application, a user would enter the number.)

```
EXEC SQL BEGIN DECLARE SECTION;
struct customer_type /* host variables */
{
    short custnum;
    char  custname[19];
    char  street[23];
    char  city[15];
    char  state[13];
    char  postcode[11];
} customer;

int find_this_customer;
EXEC SQL END DECLARE SECTION;
...

...
void not_found_function(void) /* For NOT FOUND condition */
{
    ...
}
void find_record(void)
{
    find_this_customer = 5635;
    EXEC SQL SELECT customer.custname,
                    customer.street,
                    customer.city,
                    customer.state,
                    customer.postcode
    INTO :customer.custname,
        :customer.street,
        :customer.city,
        :customer.state,
```

```

        :customer.postcode
    FROM sales.customer
    WHERE customer.custnum = :find_this_customer
    BROWSE ACCESS;

/* Process data returned by the SELECT statement */
...
}
int main(void)
{
EXEC SQL WHENEVER NOT FOUND CALL :not_found_function;

find_record();
...
}

```

## 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 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 `find_this_employee`. The SELECT statement retrieves only one row because the primary key value is unique.

```

find_this_employee = input_empnum /* set host variable */

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;

```

## Multirow SELECT Statement

Applications frequently request a group of rows for display on a screen, then request the next sequential group of rows.

If the operation is performed in a Pathway environment, a context-free server must receive the starting position for requesting the next set of records from the requester. It cannot save the starting position from a previous operation.

Assume that the initial request from the requester passes a blank or zeros, and that each subsequent request passes the search column values of the last record returned. The server uses the values sent from the requester to establish the starting position in the table. The server fetches the next set of rows from that position.

These examples illustrate several ways to define cursors that reposition on a key value. The illustrations start with a simple solution and proceed to increasingly complex solutions.

## Simple Example

In this example, the search is performed on one column, which is the primary key of the table. For example, a cursor `SELECT` to retrieve all the columns in the `EMPLOYEE` table by primary key.

The `WHERE` clause in this example selects on a primary key value. This means that the SQL compiler can choose the primary index as the access path so that each `FETCH` statement returns the next row in primary key sequence. This code is simple and efficient:

```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, SALARY
FROM =EMPLOYEE
WHERE EMPNUM > :LASTEMPNUM
ORDER BY EMPNUM
```

## A More Complex Example

In a slightly more complex example, suppose that the search uses a column that is not the primary key (for example, the column `LAST_NAME`). In this case, the query should be faster if there is an index on `LAST_NAME`. Suppose that there is an index on `LAST_NAME` in this example:

```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, SALARY
FROM =EMPLOYEE
WHERE LAST_NAME > :LAST-LNAME
ORDER BY LAST_NAME
```

When an index on a nonkey column is efficient and available, the SQL compiler probably chooses that index.

## The Most Complex Example

A more complex problem occurs when the key is composed of multiple columns. In this case, you should generally use a multivalue predicate for the comparison. This type of predicate compares multiple columns with multiple values.

Suppose that you want to retrieve the next row in sequence by last name and first name, and an index exists on the two columns containing the last name and the first name. Code this type of request by using a multivalue predicate. A multivalue predicate allows you to concatenate two or more columns and compare them with two or more concatenated values. This type of predicate retrieves the next name in sequence. For example:

```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, SALARY
FROM =EMPLOYEE
WHERE (LAST_NAME, FIRST_NAME) > :LAST_NAME, :FIRST_NAME
ORDER BY LAST_NAME, FIRST_NAME
```

If there is an index on the two columns `LAST_NAME` and `FIRST_NAME` in that order, this query is probably as efficient as it can be.

Do not code this request with this WHERE clause:

```
WHERE LAST_NAME > :LAST-LNAME
AND FIRST_NAME > :LAST-FNAME
```

This clause does not retrieve names with the same last name as :LAST-LNAME and a first name greater than :LAST-FNAME.

Also, do not code this request with this WHERE clause:

```
WHERE ( (LAST_NAME = :LAST-LNAME
AND FIRST_NAME > :LAST-FNAME)
OR LAST_NAME > :LAST-LNAME)
```

This clause would produce the correct results, but very slowly. Whenever possible, avoid the OR disjunctive in a WHERE clause.

## 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 runs an INSERT statement to transfer these host variable values to the table.

To run 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 that you include in a SELECT statement. For details, see [Required Access Authority](#) on page 7-1.

NonStop SQL/MP returns these values to `sqlcode` after an INSERT statement.

<b>sqlcode Value</b>	<b>Description</b>
0	The INSERT statement was successful.
100	No rows qualified for an insert using a SELECT statement specification.
<0	An error occurred; <code>sqlcode</code> contains the error number.
>0 (!100)	A warning occurred; <code>sqlcode</code> contains the first warning number.

If an INSERT statement runs 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, use the SQLCADISPLAY or SQLCATOBUFFER 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;
short hv_jobcode;           /* host variables */
char hv_jobdesc[18];
...
EXEC SQL END DECLARE SECTION;
...
void insert_job(void)
{
/* Set the values of hv_jobcode and hv_jobdesc */
...

EXEC SQL INSERT INTO persnl.job
              (jobcode, jobdesc)
              VALUES (:hv_jobcode, :hv_jobdesc) ;

...
}
```

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:

```
/* Variable declarations: */
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE persnl.employee AS emp_tbl;
struct emp_tbl emp;
...
short ind_1;
EXEC SQL END DECLARE SECTION;
...
/* Executable statements: */
ind_1 = -1;
EXEC SQL INSERT INTO persnl.employee
              VALUES (:emp.empnum, :emp.first_name,
                      :emp.last_name, :emp.deptnum, :emp.jobcode,
                      :emp.salary INDICATOR :ind_1);
```

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

```
EXEC SQL INSERT INTO persnl.employee
              VALUES (:emp.empnum, :emp.first_name,
                      :emp.last_name, :emp.deptnum, :emp.jobcode,
                      NULL);
```

## Inserting a Timestamp Value

This example inserts a timestamp value into `tablet.columna`. The `columna` definition specifies the data type `TIMESTAMP DEFAULT CURRENT`. The example uses the `JULIANTIMESTAMP` system procedures and the SQL `CONVERTTIMESTAMP` function. To call system procedures, a program must include declarations from the `cextdecs` header file.

```
#include <cextdecs(JULIANTIMESTAMP)>
...
EXEC SQL BEGIN DECLARE SECTION;
    long long dtvar;
EXEC SQL END DECLARE SECTION;
short sqlcode;

int main(void)
{
    ...
    /* Get Julian timestamp in GMT: */
    dtvar = JULIANTIMESTAMP();
    EXEC SQL BEGIN WORK;
    /* Insert value into tablet: */
    EXEC SQL INSERT INTO tablet (columna)
        VALUES (CONVERTTIMESTAMP (:dtvar));
    EXEC SQL COMMIT WORK;
    ...
}
```

## UPDATE Statement

The `UPDATE` statement updates the values in one or more columns in a single row or in a set of rows of a table or protection view.

To update a set of rows, one row at a time using a cursor, see [Using SQL Cursors](#) on page 4-14.

To run 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 tables or views specified in subqueries of the search condition. For details, see [Required Access Authority](#) on page 7-1.

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

NonStop SQL/MP returns these values to `sqlcode` after an UPDATE statement.

<b>sqlcode Value</b>	<b>Description</b>
0	The UPDATE statement was successful.
100	No rows were found on a search condition.
<0	An error occurred; <code>sqlcode</code> contains the error number.
>0 (!100)	A warning occurred; <code>sqlcode</code> contains the first warning number.

The UPDATE statement updates rows in sequence. If an error occurs, NonStop SQL/MP returns an error code to `sqlcode` and terminates the UPDATE operation. The SQLCA structure contains the number of rows that have been updated. (If the UPDATE statement fails, do not rely on the SQLCA structure for an accurate count of the number of updated rows.) To return the contents of the SQLCA structure, use the SQLCADISPLAY or SQLCATOBUFFER 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`.

```
EXEC SQL BEGIN DECLARE SECTION;
struct orders_type
{
    long  ordernum;
    long  order_date;
    long  deliv_date;
    short salesrep;
    short custnum;
} orders;

long newdate;
EXEC SQL END DECLARE SECTION;
...

...
void update_orders(void)
{
    ...

    newdate = update_date;
    orders.ordernum = update_ordernum;

    EXEC SQL UPDATE sales.orders SET deliv_date = :newdate
        WHERE ordernum = :orders.ordernum
        STABLE ACCESS;

    ...
}
```

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).

## 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;
```

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 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;
```

## Updating Columns With Null Values

This example updates the specified SALARY column to a null value using an indicator variable. The set\_to\_nulls host variable specifies the row to update.

```
/* indicator-var is set to -1 */
EXEC SQL UPDATE persnl.employee
  SET SALARY = :emp_tbl.salary
  INDICATOR :indicator_var
  WHERE :emp_tbl.jobcode = set_to_nulls;
```

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

```
EXEC SQL UPDATE persnl.employee SET SALARY = NULL
  WHERE :emp_tbl.jobcode = set_to_nulls;
```

## 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 using a cursor, see [Using SQL Cursors](#) on page 4-14.)



To run a DELETE statement, a process started by the program must have read and write access to the table or view and to tables or views specified in subqueries of the search condition. For details, see [Required Access Authority](#) on page 7-1.

NonStop SQL/MP returns these values to `sqlcode` after a DELETE statement.

<b>sqlcode Value</b>	<b>Description</b>
0	The DELETE statement was successful.
100	No rows were found on a search condition.
<0	An error occurred; <code>sqlcode</code> contains the error number.
>0 (!100)	A warning occurred; <code>sqlcode</code> contains the first warning number.

After a successful DELETE operation, the SQLCA structure contains the number of rows deleted. If an error occurs, 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 example 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 `hostvar_empnum`.

```
EXEC SQL DELETE FROM persnl.employee
      WHERE empnum = :hostvar_empnum;
```

## 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 ;
```

This example deletes all suppliers from the PARTSUPP table who charge more than `terminal_max_cost` for a terminal. Terminal part numbers range from `term_first_num` to `term_last_num`.

```
EXEC SQL DELETE FROM invent.partsupp
      WHERE partnum BETWEEN :term_first_num AND :term_last_num
      AND partcost > :terminal_max_cost ;
```

# 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.

[Example 4-1](#) shows the steps that you follow to declare and use a static SQL cursor in a C program. A cursor operation must run each statement in this specified order. All steps are required, even if you run the FETCH statement only once to retrieve a single row.

---

## Example 4-1. Using a Static SQL Cursor in a C Program

```

/* C source file */

1 EXEC SQL BEGIN DECLARE SECTION ;
  ... /* Declare host variable(s). */
EXEC SQL END DECLARE SECTION ;
  ...

2 EXEC SQL DECLARE cursor1 CURSOR FOR
      SELECT column1, column2, column    n
      FROM =table
      WHERE column1 = :hostvar_find_row ;

  ...
void find_row(void)
{
3   ...
4   hostvar_find_row = initial_value ; /* Initialize the host variable(s). */
  ...
EXEC SQL OPEN cursor1 ; /* Open the cursor. */

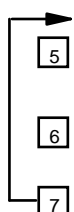
5 /* Fetch data from a row into the host variable(s). */
EXEC SQL FETCH cursor1
6     INTO :hostvar_1, :hostvar_2, :hostvar    n ;

7   ... /* Process the row values in the host variable(s). */

8   ... /* Branch back to fetch another row. */

EXEC SQL CLOSE cursor1 ; /* Close the cursor. */
}

```



The diagram shows a vertical line on the left side of the code block. A horizontal arrow points from the right side of step 8 to the right side of step 5, indicating a loop back to the fetch statement.

CDT 009CDD

The SQL statements used in [Example 4-1](#) are described in detail later in this section:

- [DECLARE CURSOR Statement](#) on page 4-18
- [OPEN Statement](#) on page 4-19
- [FETCH Statement](#) on page 4-20
- [Multirow SELECT Statement](#) on page 4-21
- [UPDATE Statement](#) on page 4-22
- [Multirow DELETE Statement](#) on page 4-23
- [CLOSE Statement](#) on page 4-24

For information about declaring host variables, see [Section 2, Host Variables](#).

## Steps for Using a Cursor

1. Declare any host variables you plan to use with the cursor.
2. Name and define the cursor 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 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 run within the same TMF transaction as the OPEN statement.

6. Process the column values returned from the current row to 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.)

## Process Access ID (PAID) Requirements

To use an SQL cursor, a process started by the program must have the access authority shown in this table. NonStop SQL/MP checks this authority when the program opens the cursor. For details, see [Required Access Authority](#) on page 7-1.

### 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. NonStop SQL/MP tests the table only for read access when the OPEN statement runs. However, because a DELETE operation requires write access, NonStop SQL/MP checks for write access when you run the DELETE statement.

A program contending for data access with other users can specify the IN EXCLUSIVE MODE clause in the associated SELECT statement. NonStop 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.

## Cursor Position

[Table 4-2](#) describes the SQL statements that affect the cursor position in a program. The cursor position is similar to the record position in a sequential file.

---

**Table 4-2. Determining the Cursor Position**

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 NonStop SQL/MP to guarantee cursor stability, you must 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 C program requires no special authorization to run 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, NonStop 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 the SQL statements that refer to them.
- Place the DECLARE CURSOR statement in listing order before 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 statements that reference the cursor.
- The DECLARE CURSOR statement does not affect the values in the SQLCA and SQLSA data structures.

This example declares a cursor `list_by_partnum`:

```
EXEC SQL BEGIN DECLARE SECTION;
struct parts_type          /* host variables */
{
    short partnum;
    char  partdesc[19];
    long  price;
    short qty_available
} parts_rec;
EXEC SQL END DECLARE SECTION;
...

EXEC SQL DECLARE list_by_partnum CURSOR FOR
    SELECT partnum,
           partdesc,
           price,
           qty_available
    FROM   =parts
    WHERE  partnum >= :parts_rec.partnum
    ORDER BY partnum
    BROWSE ACCESS;
...
```

## 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 run an OPEN statement for a cursor, a process started by the program must have the access authority described in [Process Access ID \(PAID\) Requirements](#) on page 4-16. For details, see [Required Access Authority](#) on page 7-1.

If the associated SELECT statement contains host variables in the WHERE clause, you must initialize these host variables before you run the OPEN statement. When the OPEN statement runs, NonStop 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 run 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 run before any FETCH statements for the cursor.
- For audited tables and views, the OPEN statement must run within a TMF transaction.
- If data is materialized by the OPEN operation, NonStop 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;
...
```

## 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 run 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.

NonStop SQL/MP returns these values to `sqlcode` after a FETCH statement.

<b>sqlcode Value</b>	<b>Description</b>
0	The FETCH statement was successful.
100	The end of a table was encountered.
<0	An error occurred; <code>sqlcode</code> contains the error number.
>0 (!100)	A warning occurred; <code>sqlcode</code> contains the first warning number.

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

NonStop SQL/MP resets values in an SQLSA structure immediately before a FETCH statement runs. If you use an SQLSA value elsewhere in your program, save the value in a variable immediately after the FETCH statement runs. 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 runs.

For audited tables and views, the FETCH statement must run 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;

struct parts_type      /* host variables */
{
    short partnum;
    char  partdesc[19];
    long  price;
    short qty_available
} parts_rec;
...
EXEC SQL END DECLARE SECTION;
...

...
EXEC SQL DECLARE list_by_partnum CURSOR FOR
        SELECT partnum,partdesc,price,qty_available
        FROM    =parts
```



```

        WHERE partnum >= :parts_rec.partnum
        ORDER BY partnum
        BROWSE ACCESS;
...

void list_func(void)
{
EXEC SQL OPEN list_by_partnum;
EXEC SQL FETCH list_by_partnum
        INTO :parts_rec.partnum,
            :parts_rec.partdesc,
            :parts_rec.price,
            :parts_rec.qty_available;
...
}

```

## 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 run 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.

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;
char begin_code[11], end_code[11];
EXEC SQL INVOKE =customer AS customer_struct;
struct customer_struct customer_row;
...
EXEC SQL END DECLARE SECTION;

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;
...
void list_customers(void)
{

```

```

...
EXEC SQL OPEN get_name_address;

... /* Set values for begin_code and end_code. */

EXEC SQL FETCH get_name_address
      INTO :customer_row.custname,
           :customer_row.street,
           :customer_row.city,
           :customer_row.state
           :customer_row.postcode;

... /* Process the row values. */

EXEC SQL CLOSE get_name_address;
}

```

## 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 run an UPDATE statement, a process started by the program must have read and write access to the table or view being updated. This process 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 run within the same TMF transaction as the OPEN and FETCH statements for the cursor.

This example uses the cursor `get_by_partnum` and host variables `new_partdesc`, `new_price`, and `new_qty` to update the PARTS table:

```

EXEC SQL DECLARE get_by_partnum CURSOR FOR
      SELECT parts.partnum,
             parts.partdesc,
             parts.price,
             parts.qty_available
      FROM   sales.parts
      WHERE  (parts.partnum >= :parts.partnum )
      STABLE ACCESS
      FOR UPDATE OF parts.partdesc,
                    parts.price,
                    parts.qty_available;

```

```

...
EXEC SQL OPEN get_by_partnum;
... /* Set values of the host variables. */

EXEC SQL FETCH get_by_partnum INTO ... ;

... /* Test the value(s) in the current row. */

/* Update the current row */
EXEC SQL UPDATE sales.parts
    SET parts.partdesc      = :new_partdesc,
        parts.price        = :new_price,
        parts.qty_available = :new_qty
    WHERE CURRENT OF get_by_partnum;

... /* Branch back to FETCH to get the next row. */
EXEC SQL CLOSE get_by_partnum;

```

## 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 by a DROP TABLE statement.

To run a DELETE statement, a process started by the program must have read and write access to the table or view and to tables or views specified in subqueries of the search condition. For more information about process access, see [Required Access Authority](#) on page 7-1.

A DELETE statement must run 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 run 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 `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 >= :parts.partnum);

...

EXEC SQL OPEN get_by_partnum;

EXEC SQL FETCH get_by_partnum ... ;

... /* Test the value(s) in the current row. */

/* Delete the current row */
EXEC SQL DELETE FROM sales.parts
        WHERE CURRENT OF get_by_partnum ;
... /* Branch back to FETCH the next row. */
EXEC SQL CLOSE get_by_partnum;

```

## CLOSE Statement

The CLOSE statement closes an open SQL cursor. After the CLOSE statement runs, 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 run 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:

```

...

void list_func(void)
{
...
EXEC SQL CLOSE list_by_partnum;
}

```

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. Only dynamic cursors can be foreign cursors. Static cursors cannot be foreign cursors.

A reference to a foreign cursor contains two parts, a procedure name and a cursor name. This example references a foreign cursor, `list_by_partnum`, which is declared in the procedure `update_inv`:

```
update_inv.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 compile source file. References to a dynamic foreign cursor are resolved at run time by the SQL Executor.

The cursor declaration and the PREPARE statement 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 and to maintain proper association between a procedure and a particular statement name.

This example declares a cursor `list_by_partnum`:

```
update_inv(void)
{
EXEC SQL BEGIN DECLARE SECTION;
struct parts_type
{
/* define host variables here */
} parts_rec;
EXEC SQL END DECLARE SECTION;
...

EXEC SQL DECLARE list_by_partnum CURSOR FOR
      SELECT partnum, /* defined above */
             partdesc,
             price,
             qty_available
      FROM   =parts
      WHERE  partnum >= :parts_rec.partnum
      ORDER BY partnum
      BROWSE ACCESS;
EXEC SQL PREPARE dynamic_statement FROM :hv_text;
}
```

These statements open, fetch, and close a foreign cursor `list_by_partnum` that are declared in the procedure `update_inv`:

```
/* Loop while not EOF: */
void update_inv_total(void)
{
/* describe input and output here */
exec sql open update_inv.list_by_partnum using descriptor input-
sqllda;
}
{
/* describe input and output here */
exec sql fetch update_inv.list_by_partnum using descriptor
output-sqllda;
```

```
}  
}  
/* describe input and output here */  
exec sql close update_inv.list_by_partnum;  
}
```

[Table 5-1](#) describes the NonStop SQL/MP system procedures, which are written in TAL, that a C program can call to return various SQL information. These procedures are listed alphabetically.

---

**Table 5-1. SQL/MP System Procedures**

Procedure	Description
<b>To Return Error and Warning Information</b>	
SQLCADISPLAY	Writes to a file or terminal the error and warning messages that NonStop SQL/MP returns to the SQLCA structure.
SQLCAFSCODE	Returns information about file-system, disk-process, or operating system errors from the SQLCA structure.
SQLCAGETINFOLIST	Returns to an area in the program a specified subset of the error or warning information in the SQLCA structure.
SQLCATOBUFFER	Returns to a record area in the program the error or warning messages that NonStop SQL/MP returns to the SQLCA structure.
<b>To Return Version Information</b>	
SQLGETCATALOGVERSION	Returns the version of an SQL catalog.
SQLGETOBJECTVERSION	Returns the version of an SQL object (table, index, or view).
SQLGETSYSTEMVERSION	Returns the version of the SQL file-system and disk-process components for a specified system.
<b>To Return Execution Statistics</b>	
SQLSADISPLAY	Writes to a file or terminal the execution statistics that NonStop SQL/MP returns to the SQLSA structure.

---

# Guardian System Procedures

In addition to the procedures in [Table 5-1](#) on page 5-1, a C program can also call the Guardian 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 name.
FILE_GETINFOLIST__	Returns detailed information about a file using the file number. Item codes 40, 82, 83, 84, and 85 apply to NonStop SQL/MP.
FILE_GETINFOLISTBYNAME_	Returns detailed information about a file using the file name. Item codes 40, 82, 83, 84, and 85 apply to NonStop SQL/MP.

---

## cextdecs Header File

The `cextdecs` header file contains source declarations for the SQL/MP and Guardian system procedures. Use the `#include` directive as shown in this example to copy the declarations from the `cextdecs` header file for the procedures you want to call in your program:

```
...
#include <cextdecs ( FILE_OPEN_,      \
                   READ,              \
                   WRITEREAD,         \
                   FILE_GETINFO_,     \
                   FILE_CLOSE_,       \
                   SQLCADISPLAY,      \
                   SQLCAFSCODE )> nolist
...
```

## 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 C program opens and reads the SQL message file when it calls an SQL system procedure that returns error or status information (for example, `SQLCADISPLAY` or `SQLCATOBUFFER`).

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_node` `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_node` DEFINE either interactively from TACL or SQLCI, or programmatically from a C 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_node` 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, NonStop 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 C 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*.

## SQLCADISPLAY

The SQLCADISPLAY procedure displays error or warning information that NonStop SQL/MP returns to the SQLCA data structure. SQLCADISPLAY writes this information to a file or terminal.

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

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

NonStop 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 for display, call the SQLCADISPLAY procedure to convert this information to an appropriate format.

```

#include <cextdecs(SQLCADISPLAY)>

void SQLCADISPLAY (
    short *sqlca,                /* i */
    [ short output_file_number,  ] /* i */
    [ short output_record_length, ] /* i */
    [ short *sql_msg_file_number, ] /* i:o */
    [ short errors,              ] /* i */
    [ short warnings,            ] /* i */
    [ short statistics,          ] /* i */
    [ short caller_error_loc,     ] /* i */
    [ short internal_error_loc,   ] /* i */
    [ char *prefix,               ] /* i */
    [ short prefix_length,        ] /* i */
    [ char *suffix,               ] /* i */
    [ short suffix_length,        ] /* i */
    [ short *detail_params       ] /* i */
);

```

*sqlca*

is a pointer to the SQLCA structure. The C compiler automatically declares the SQLCA structure when you specify the INCLUDE SQLCA directive.

*output\_file\_number*

is the output file number. If you omit this value or set it to a negative value, SQLCADISPLAY displays information at your home terminal. In this case, SQLCADISPLAY opens your home terminal, displays the message, and then closes your terminal. This parameter is ignored if *detail\_params* specifies sequential I/O (SIO).

*output\_record\_length*

is the length in bytes of records to be written to the output file. The length must be an integer value from 60 through 600. The default length is 79 bytes.

*sql\_msg\_file\_number*

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 SQLCADISPLAY (or the SQLCATOBUFFER 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_node` DEFINE. For more information, see [SQL Message File](#) on page 5-2.

#### *errors*

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*

controls the display of warning messages:

Y    Display all warning messages.

N    Display all warning messages.

B    Display all warnings but suppress this prefix:

      WARNING from *subsystem* [nn]

The default is Y.

#### *statistics*

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*

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*

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*

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

*prefix\_length*

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

*suffix*

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

*suffix\_length*

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

*detail\_params*

determines whether the program uses sequential I/O (SIO) or Enscribe I/O to write to the output file. The parameter *detail\_params* points to a structure with this layout:

```
struct detail_params_type
{
    char sio;
    short *out_fcb_1;
    short *out_fcb_2;
} detail_params;
```

*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.

The default is Enscribe I/O.

Additional considerations for the SQLCADISPLAY procedure are:

- NonStop SQL/MP returns errors as negative numbers and warnings as positive numbers. Therefore, you might accordingly need to modify your program.
- If there is no text for an error number, NonStop SQL/MP displays this message:  
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 wrapped at word boundaries producing subsequent lines indented 5 spaces.
- The SQLCA can contain a maximum of 7 errors and 180 bytes of text of the actual parameters returned to the program. Any information that exceeds these limits is lost. SQLCADISPLAY displays a warning message that indicates when information is lost.

This example calls the SQLCADISPLAY procedure using default parameters:

```
#include <cextdecs (SQLCADISPLAY)>
...
/* Variable declarations: */
...
EXEC SQL INCLUDE STRUCTURES SQLCA VERSION 300;
EXEC SQL INCLUDE SQLCA;
...

/* Error handling function: */
...
void handle_errors(void)
{
    SQLCADISPLAY( (short *) &sqlca);
}
...
```

This example shows diagnostic messages the SQLCADISPLAY procedure might generate:

```
*** WARNING from SQL [100]: Record not found or end of file
*** encountered on table \SYS1.$VOL1.SALES.ODETAIL.

*** SQLCA display of SQL statement at: SCAN.#201.1 process
\SYS1.$B
*** Error detected within SQL executor at: EXE_EXEC.#450
*** ERROR from SQL [-8408]: Division by zero occurred.
```

```
*** Statistics: Rows accessed/affected: 10
*** Estimated cost: 100
```

## 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 the operating system. If there was no such error, SQLCAFSCODE returns 0. If the SQLCA is full when an error occurs, the error is lost.

```
#include <cextdecs(SQLCAFSCODE)>

short SQLCAFSCODE (
    short *sqlca,      /* i */
    [ short first_flg ] /* i */
);
```

*sqlca*

is a pointer to the SQLCA structure. The C compiler declares the SQLCA structure if you specify the INCLUDE SQLCA directive.

*first\_flg*

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.

This example calls the SQLCAFSCODE procedure:

```
#include <cextdecs(SQLCAFSCODE)>
...
short fserr;

EXEC SQL INCLUDE SQLCA;

...
fserr = SQLCAFSCODE ((short *) &sqlca);
...
```

# SQLCAGETINFOLIST

The SQLCAGETINFOLIST procedure returns error or warning information that NonStop SQL/MP sets in the SQLCA structure. You specify a list of numbers, called item codes (shown in [Table 5-4](#) on page 5-11), to specify the error or warning information, and SQLCAGETINFOLIST returns the information to a structure in your program.

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

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

SQLCAGETINFOLIST returns zero after a successful operation or one of the error codes shown in [Table 5-3](#) on page 5-11 if an error occurs.

---

**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.

---

```
#include <cextdecs(SQLCAGETINFOLIST)>

short SQLCAGETINFOLIST (
    short *sqlca,           /* i */
    short *item_list,       /* i */
    short number_items,     /* i */
    short *result,          /* o */
    short result_max,       /* i */
    [ short error_index, ]  /* i */
    [ short names_max, ]   /* i */
    [ short params_max, ]  /* i */
    [ short *result_len, ] /* o */
    [ short *error_item ]  /* o */
);
```

*sqlca*

is a pointer to the SQLCA structure. The C compiler automatically declares the SQLCA structure if you specify the INCLUDE SQLCA directive.

*item\_list*

is an array of item codes that describes the information you want returned in the *result* structure. For a list of these item codes, see [Table 5-4](#) on page 5-11.

*number\_items*

is the number of items you specified in the *item\_list* array.

*result*

is a structure 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*

is the maximum size, in bytes, of the *result* structure.

*error\_index*

is the index of the SQLCA error or warning entry.

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. NonStop SQL/MP uses *error\_index* to access this table to determine the error or warning.

If *error\_index* is omitted, NonStop SQL/MP returns the first error record.

*names\_max*

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

*params\_max*

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

*result\_len*

is the total number of bytes used in the *result* structure.

*error\_item*

is the index of the item being processed when the error occurred. The index starts at 0 (zero).



[Table 5-3](#) lists the SQLCAGETINFOLIST error codes.

**Table 5-3. SQLCAGETINFOLIST Procedure Error Codes**

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 <i>names_max</i> parameter less than or equal to zero.
8517	The program specified a <i>params_max</i> parameter less than or equal to zero.

[Table 5-4](#) lists the codes you can specify in the *item\_list* array.

**Table 5-4. SQLCAGETINFOLIST Procedure Item Codes** (page 1 of 2)

Item Code	Size (Bytes)	Description
1	2	Version of the SQLCA structure.
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.
10	2	Syntax error location. If there was no syntax error, SQL returns -1.

**Table 5-4. SQLCAGETINFOLIST Procedure Item Codes** (page 2 of 2)

Item Code	Size (Bytes)	Description
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.  NonStop SQL/MP returns -1 if there are no parameters.
26	2	Number of parameters for this error.
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.

In [Example 5-1](#), the SQLCAGETINFOLIST procedure returns the name of the function containing the SQL statement that produced one or more errors or warnings, the name length of the function, and the number of errors or warnings. To avoid coding the maximum length for the function name (`err_warn.name_len` in the example), call SQLCAGETINFOLIST with item code 7 (the actual length of the function name) and then call SQLCAGETINFOLIST again with a buffer of that size.

---

### Example 5-1. Example of the SQLCAGETINFOLIST Procedure

```
#include <cextdecs(SQLCAGETINFOLIST)>

...
#define MAX_NAME_LEN 30
#define ITEM_LIST_SIZE 3
struct    /* structure to hold error information */
{
    short name_len;
    short num_errs;
    char name[MAX_NAME_LEN];
} err_warn;

...
EXEC SQL INCLUDE SQLCA; /* include SQLCA structure */
short error_code;       /* variable to hold return code */
/* Declare and initialize the item-list array */
short item_list[ITEM_LIST_SIZE]
    = { 7, /* code for name length */
        3, /* code for no of errors/warnings */
        8 }; /* code for procedure ID */

...
error_code = SQLCAGETINFOLIST
    ( (short *) &sqlca, /* SQLCA structure */
      item_list, /* list of item codes */
      item_list_size, /* number of items */
      (short*) &err_warn, /* result area */
      sizeof err_warn, /* size of result area */
      , /* no error index needed */
      max_name_len ); /* Truncate names > 30 */

...

```

---

# SQLCATOBUFFER

The SQLCATOBUFFER procedure writes to a buffer the error or warning messages that NonStop SQL/MP returns to the program. This buffer is a structure declared in variable declarations in the program.

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

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

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

```
#include <cextdecs(SQLCATOBUFFER)>

void SQLCATOBUFFER (
    short *sqlca                                /* i */
    char *output_buffer                         /* i:o */
    short output_buffer_length                 /* i */
    [ short first_record_number ]             /* i */
    [ short *output_records ]                 /* o */
    [ short *more ]                           /* o */
    [ short output_record_length ]            /* i */
    [ short *sql_msg_file_number ]            /* i:o */
    [ short errors ]                          /* i */
    [ short warnings ]                        /* i */
    [ short statistics ]                     /* i */
    [ short caller_error_loc ]                /* i */
    [ short internal_error_loc ]              /* i */
    [ char *prefix ]                          /* i */
    [ short prefix_length ]                   /* i */
    [ char *suffix ]                          /* i */
    [ short suffix_length ]                   /* i */
);
```

*sqlca*

is a pointer to the SQLCA structure. The C compiler automatically declares the SQLCA structure when you specify the INCLUDE SQLCA directive.

*output\_buffer*

is the name of the buffer where SQLCATOBUFFER writes the error information.

*output\_buffer\_length*

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*

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

The default is 1.

The count of lines begins with 1. To obtain more than one error record, increment the value in *first\_record\_number*.

*output\_records*

is the number of records (lines) written to *output\_buffer*.

*more*

is a flag that indicates whether all desired lines fit into the *output\_buffer*:

Y     There were additional records; the buffer overflowed.

N     There were no additional records.

*output\_record\_length*

defines the length of records to be written to the *output\_buffer*. The length must be an integer value from 60 through 600. The default is 79 bytes.

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

*sql\_msg\_file\_number*

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 SQLCATOBUFFER (or 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_node` DEFINE. For more information, see [SQL Message File](#) on page 5-2.

*errors*

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*

controls the writing of warning messages to the buffer:

- Y Write all warning messages.
- N Write all warning messages.
- B Write all warnings but suppress this prefix:

WARNING from *subsystem* [*nn*]

The default is Y.

*statistics*

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*

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*

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*

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

*prefix\_length*

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

*suffix*

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

*suffix\_length*

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

Additional considerations for the SQLCATOBUFFER procedure are:

- NonStop 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, NonStop SQL/MP displays this message:

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.

- The SQLCATOBUFFER procedure starts 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. SQLCATOBUFFER returns to *output\_records* a count of the lines moved to the buffer. If an overflow occurs, the procedure sets the *more* flag to Y.
- On an overflow condition, your program can retrieve the remainder of the error message text by calling SQLCATOBUFFER again and setting *first\_record\_number* to *output\_records* + 1.

In this example, the SQLCAFSCODE procedure writes the error or warning messages to `sql_msg_buffer`, a buffer declared as 600 bytes:

```
#include <cextdecs(SQLCATOBUFFER)>
...
EXEC SQL INCLUDE SQLCA;
...
{
    char sql_msg_buffer[600];
    ...
    SQLCATOBUFFER ((short *) &sqlca, sql_msg_buffer, 600);
}
...
```

## SQLGETCATALOGVERSION

The SQLGETCATALOGVERSION procedure returns the version of a catalog.

SQLGETCATALOGVERSION returns zero after a successful operation or a nonzero value to indicate an error or warning condition. For a description of SQL errors, see the *SQL/MP Messages Manual*.

```
#include <cextdecs(SQLGETCATALOGVERSION)>

short SQLGETCATALOGVERSION (
                                [ char *catalog_name ] ,    /* i */
                                short *sql_version           /* o */
);
```

*catalog\_name*

is the fully qualified 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*

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

---

**Note.** Although version 340 SQL/MP software supports the SQLGETCATALOGVERSION procedure, HP might not support this procedure in a future RVU. 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.

SQLGETOBJECTVERSION returns zero after a successful operation or a nonzero value to indicate an error or warning condition. For a description of SQL errors, see the *SQL/MP Messages Manual*.

```
#include <cextdecs(SQLGETOBJECTVERSION)>

short SQLGETOBJECTVERSION (
    char *object_name ,    /* i */
    short *sql_version    /* o */
);
```

*object\_name*

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

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

*sql\_version*

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

---

**Note.** Although version 340 SQL/MP software supports the SQLGETOBJECTVERSION procedure, HP might not support this procedure in a future RVU. 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 SQL/MP file system and disk process components running on a system. For a specific node, assume that all SQL/MP components are of the same PVU.

SQLGETSYSTEMVERSION returns zero after a successful operation or a nonzero value to indicate an error or warning condition. For a description of SQL errors, see the *SQL/MP Messages Manual*.

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

```
#include <cextdecs (SQLGETSYSTEMVERSION) >

short SQLGETSYSTEMVERSION (
                                [ short node_number ] , /* i */
                                short *sql_version      /* o */
                                );
```

*node\_number*

is the node number of the system for which you are requesting information. The default is the local system.

*sql\_version*

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

---

**Note.** Although version 340 SQL/MP software supports the SQLGETSYSTEMVERSION procedure, HP might not support this procedure in a future RVU. 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.

```
#include <cextdecs (SQLSADISPLAY) >

void SQLSADISPLAY (
    short *sqlsa, /* i */
    [ short *sqlca, /* i */
    [ short output_file_number , /* i */
    [ short *detail_params ] /* i */
    );
```

*sqlsa*

is a pointer to the SQLSA structure. The C compiler automatically declares the SQLSA structure when you specify the INCLUDE SQLSA directive.

*sqlca*

is a pointer to the SQLCA structure. The SQLCA structure contains the procedure name and line number of the SQL statement that sets the SQLSA structure. The

C compiler automatically declares the SQLCA structure if you specify the INCLUDE SQLCA directive. If you omit the SQLCA name, the display does not contain the procedure name and process name of the caller.

*output\_file\_number*

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

*detail\_params*

determines whether the program uses sequential I/O (SIO) or Enscribe I/O to write to the output file. The parameter *detail\_params* points to a structure with this layout:

```
struct detail_params_type
{
    char sio;
    short *out_fcb_1;
    short *out_fcb_2;
} detail_params;
```

*sio*

specifies whether sequential I/O (SIO) is used:

Y      Use SIO; ignore *output\_file\_number*.

N      Do not use SIO; write to *output\_file\_number*.

*outfcb1*

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

*outfcb2*

specifies the second output file control block if SIO is enabled. To use *outfcb2*, assign it a value greater than 0.

If you omit *detail\_params*, Enscribe I/O is the default.

## Example of the SQLSADISPLAY Display

SQLSADISPLAY displays statistics in this format:

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

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

[Table 5-5](#) describes the elements of the SQLSADISPLAY display.

**Table 5-5. SQLSADISPLAY Procedure Display Elements**

Element	Description
<code>\system.\$vol.subvol.file</code>	The fully qualified file name of the calling program
<code>#line</code>	The line number of the calling program
<code>process cpu,pin</code>	The CPU and PIN of the calling program
Table Name	The name of each table
Records Accessed	The number of records accessed in each table (includes records examined by the disk process, file system, and SQL/MP executor)
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 5-2](#) displays the type of information found in SQLSADISPLAY. To generate this display, a program follows these steps:

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

**Example 5-2. Example of the SQLSADISPLAY Display**

SQL statistics @ \sanfran.\$system.accts.prog10.#333.2 process 12,250

Table Name	Records Accessed	Records Used	Disc Reads	Message Count	Message Bytes	Lock WE
\sanfran.\$sqlvol.accts.tab10						
	123	22	3	10	3245	
\sanfran.\$vol001.fy96.employee						
	9987231	1	99999	1	100	e
\sanfran.\$sqlvol.accts.tab20						
	1	1	0	1	100	w

# 6

## Explicit Program Compilation

This section describes the explicit compilation of a NonStop C program containing embedded SQL statements and directives in the Guardian, HP NonStop Open System Services (OSS), and PC host environments using TNS, TNS/R and TNS/E compilation tools.

---

**Note.** This section contains information about some of the following G-series development tools, which are not available on H-series systems:

- TNS/R Native C compiler
- TNS/R C++ compiler
- TNS/R C++ runtime library version 2
- NonStop SQL/MP for TNS/R C
- SQL/MP Compilation Agent
- NMCOBOL compiler
- ld
- nld
- noft
- TNS/R pTAL

Continue to use the Enterprise Toolkit or G-series servers for your G-series development tasks.

---

This section includes:

[Developing a C Program in the Guardian Environment](#) on page 6-5

[Developing a C Program in the OSS Environment](#) on page 6-28

[Developing a C Program in a PC Host Environment](#) on page 6-33

[Using CONTROL Directives](#) on page 6-34

[Using Compatible Compilation Tools](#) on page 6-36

## Explicit Program Compilation

[Table 6-1](#) and [Table 6-2](#) list the C compilers, their compilation mode, and where you can run them.

**Table 6-1. C Compilers**

Compiler	Compilation Mode	Compiler Operating Environment	NonStop System
C	TNS	Guardian	D-series G-series H-series
c89	TNS	OSS	D-series G-series
NMC	TNS/R native	Guardian	D-series G-series
c89	TNS/R native	OSS	D-series G-series
Native C cross compiler for TNS/R	TNS/R native	PC	D-series host* G-series host
CCOMP	TNS/E native	Guardian	H-series
**c89	TNS/E native	OSS	H-series
Native C cross compiler for TNS/E	TNS/E native	PC	H-series host

\* The HP Enterprise Toolkit—NonStop Edition (ETK) and PC command line are not supported on D-series systems. Instead, use the native C cross compiler of the Tandem Development Suite (TDS). For more information, see the *C/C++ Programmer's Guide for NonStop Systems*.

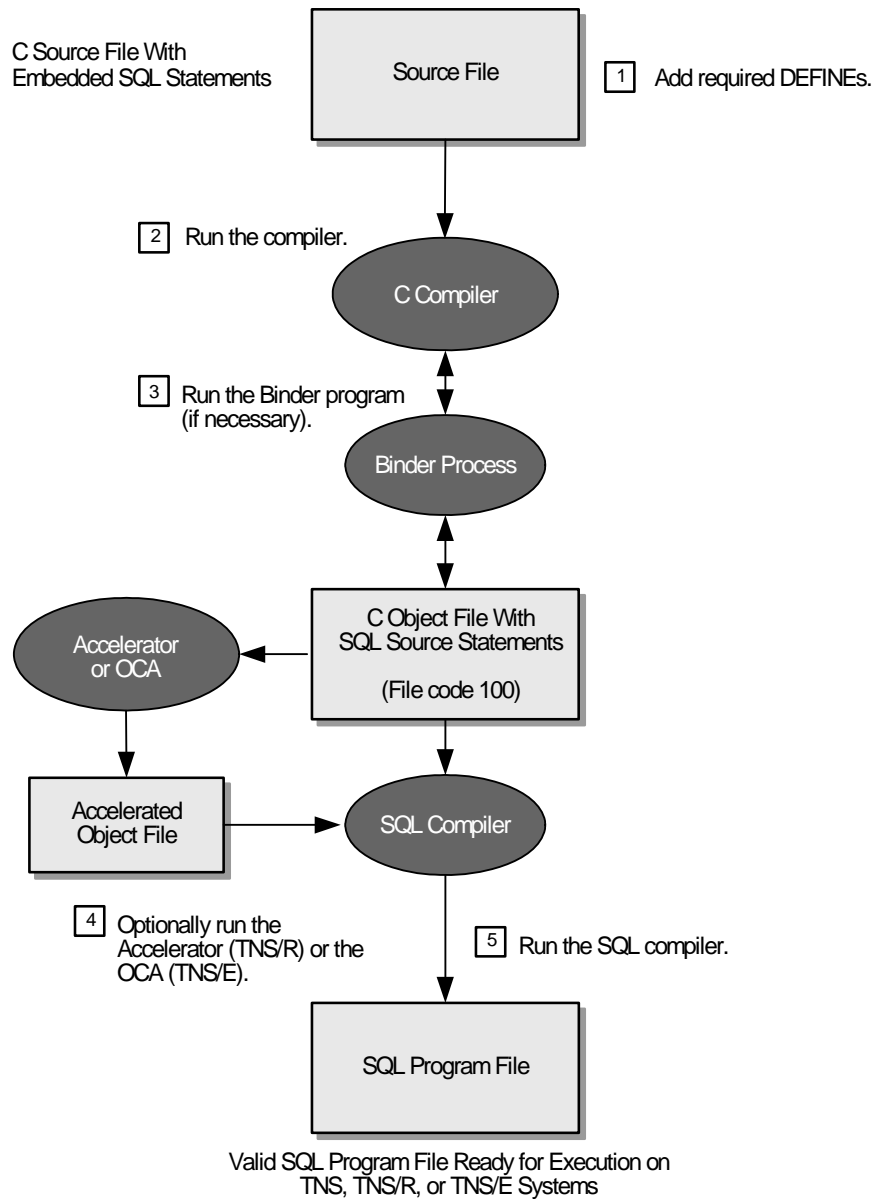
\*\* NonStop SQL/MP is only compatible with c89 and not c99.

**Table 6-2. Compilation Mode and Execution Environment**

Compilation Mode	NonStop System Where You Can Execute the Embedded SQL Program
TNS	D-series or G-series (TNS/R) H-series (TNS/E)
TNS/R native	D-series or G-series (TNS/R)
TNS/E native	H-series (TNS/E)

[Figure 6-1](#), [Figure 6-2](#) and [Figure 6-3](#) show the general steps that you follow to compile a program on the TNS, TNS/R, and TNS/E platforms.

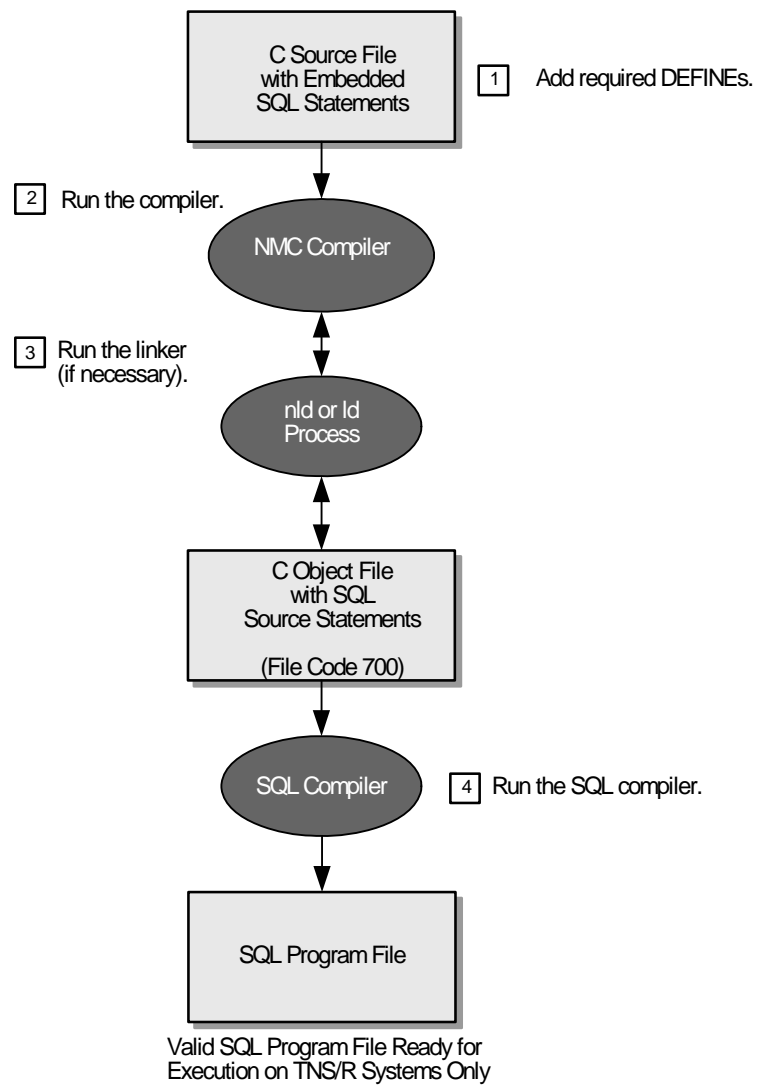
**Note.** The hardware architecture of TNS is based on complex instruction-set computing (CISC), TNS/R is based on reduced instruction-set computing (RISC) and TNS/E is based on Intel Itanium. These different architectures affect the compilation and running of SQL/MP application programs and influence where and how application development is done. For example, in a multinode environment that involves different hardware platforms (for example, TNS, TNS/R, and TNS/E), one development strategy might be to use the TNS/E platform for application development and then deploy the applications to the other platforms.

**Figure 6-1. Explicit SQL Compilation of a C Program on TNS**

In an OSS environment on a TNS/R system, Steps 2 through 5 can be invoked with the c89 utility.

VST003.vsd

---

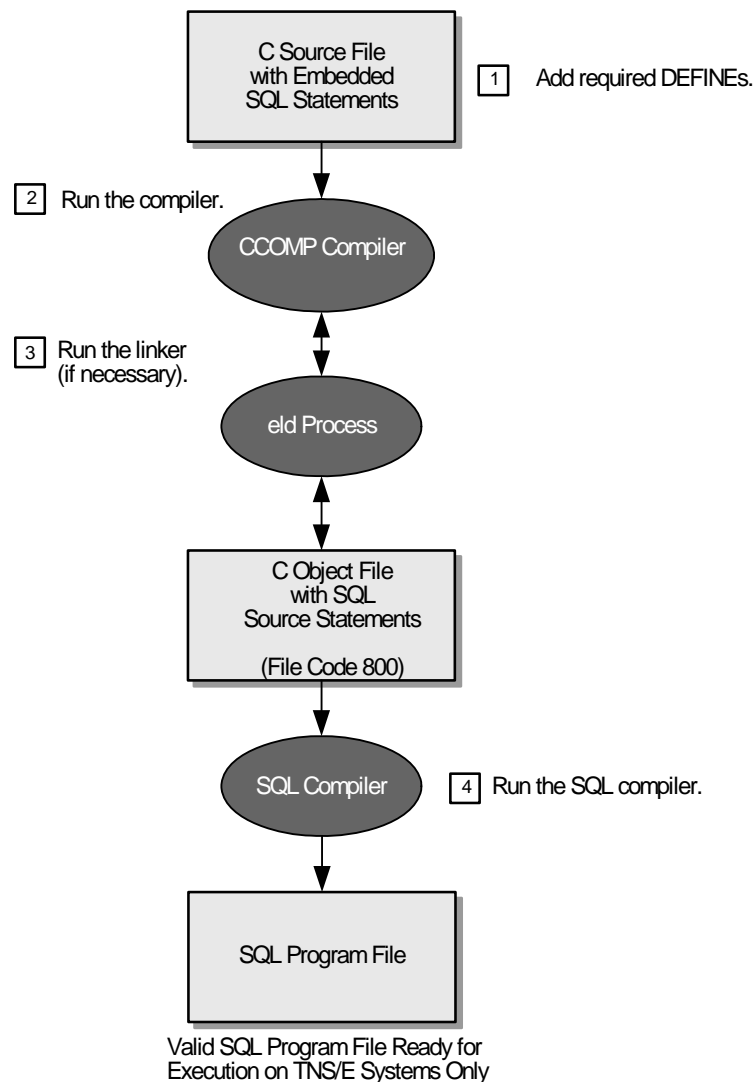
**Figure 6-2. Explicit SQL Compilation of a C Program on TNS/R**

---

In an OSS environment, Steps 2 through 4 can be invoked with the c89 utility.

VST003R.vsd



**Figure 6-3. Explicit SQL Compilation of a C Program on TNS/E**

In an OSS environment on a TNS/E system, Steps 2 through 4 can be invoked with the c89 utility.

VST003E.vsd

For information about executing an SQL program file, see [Section 7, Program Execution](#). For information about automatic SQL recompilation, see [Section 8, Program Invalidation and Automatic SQL Recompilation](#).

## Developing a C Program in the Guardian Environment

In the Guardian environment, you can develop a C program to run in either the Guardian or OSS environment.

## Using TACL DEFINES in the Guardian Environment

You can use TACL DEFINES during the compilation of a C program containing embedded SQL statements and directives:

- To use DEFINES, the TACL process DEFMODE attribute must be ON. To determine the DEFMODE setting, enter the SHOW DEFMODE command at the TACL prompt:

```
10> SHOW DEFMODE
      Defmode OFF
```

- If DEFMODE is OFF, enter a SET DEFMODE ON command:

```
11> SET DEFMODE ON
```

- Before you run the C compiler, add the DEFINES that you use for the names of tables or views 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 that you use for the names of tables, views, indexes, or collations 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
```

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 and leaves the statement uncompiled in the program file. When you run the 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 $$.#sqlist
32> SQLCOMP /IN sqlc,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. See [Running the SQL Compiler in the Guardian Environment](#) on page 6-12 for a description of the STOREDDEFINES option.

For information about using DEFINES in the OSS environment, see [Developing a C Program in the OSS Environment](#) on page 6-28.

## Specifying the SQL Pragma in the Guardian Environment

The SQL pragma indicates to the TNS C , the TNS/R NMC, or the TNS/E CCOMP compilers that a program contains embedded SQL statements or directives and specifies various options for processing the SQL statements or directives.

You can specify the SQL pragma either in your primary C source file or as a compiler option in the implicit TACL RUN command for the TNS C compiler. When you run the NMC or CCOMP compiler, you must specify the SQL pragma as a compiler option in the NMC or CCOMP command.

This example shows the SQL pragma as a TNS C compiler option:

```
C / IN csrc, OUT $S.#clist / cobj; SQL
```

To specify the SQL pragma in your C source file, use:

```
#pragma SQL [ option
              [ ( option [ , option ]... ) ]
option is:
    [ WHENEVERLIST      | NOWHENEVERLIST      ]
    [ CHAR_AS_ARRAY     | CHAR_AS_STRING    ]
    [ SQLMAP            | NOSQLMAP          ]
    [ RELEASE1          | RELEASE2          ]
    [ CPPSOURCE "filename" ]
```

WHENEVERLIST | NOWHENEVERLIST

controls the writing of active WHENEVER options to the listing file after each SQL statement is processed.

WHENEVERLIST causes the options to be written.

NOWHENEVERLIST (the default) causes the options not to be written.

CHAR\_AS\_ARRAY | CHAR\_AS\_STRING

specifies whether the INVOKE directive generates character types that contain an extra byte for the null terminator.

CHAR\_AS\_ARRAY directs INVOKE to generate character types without the extra byte.

CHAR\_AS\_STRING (the default) directs INVOKE to generate character types with the extra byte for the null terminator.

SQLMAP | NOSQLMAP

specifies whether the compiler listing includes an SQL map. This map enables you to determine SQL statements using output from the Measure program.

SQLMAP directs the C compiler to include an SQL map in the listing. An SQL map contains:

- 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.
- Source file name and number.
- Source file line number.

The table is sorted first by RTDU name and then by the SLT index number. Use this table to correlate Measure output with the SQL statement. The global RTDU contains the cursors and CONTROL directives declared in the global declarations. The SQLMAP option also directs the C compiler to include the HOSV in the compiler listing as:

```
Host Object SQL Version = 310
```

NOSQLMAP (the default) causes the SQL map not to be added.

RELEASE1 | 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.

RELEASE2 (the default) specifies version 2 features. A program that uses the RELEASE2 option is compatible with SQL/MP version 2 or version 300 (or later) software but not with version 1 software.

---

**Note.** Although the C compiler allows the use of RELEASE1 and RELEASE2 options, these options might not be supported in a future RVU. If you are using a version 300 (or later) C compiler to generate data structures, use the INCLUDE STRUCTURES directive with the VERSION 1 or VERSION 2 option.

---

CPPSOURCE "*filename*"

directs the C compiler to generate a preprocessed C source file with the name *filename*, which must be a valid Guardian file name. The generated file is empty, except for SQL source RTDUs. The CPPSOURCE option is not valid with the RELEASE1 option or a macro expansion that contains any part of an SQL statement. This option is intended primarily for use with the Distributed Workbench Facility (DWF).

## Running the TNS C Compiler in the Guardian Environment

To run the TNS C compiler in the Guardian environment, use the TACL RUN command:

```
[ RUN ] C / IN source [, OUT list-file ] [ , run-option ]...  
/  
      [ object ] [ ; compiler-option [ , compiler-option ]...  
]
```

### *source*

is the primary source file of the compilation unit. *source* can be a text disk file (code 101), a C-format disk file (code 180), a terminal, a magnetic tape unit, or a process. The default is your home terminal if your TACL process is running in interactive mode.

Do not use the same name for different functions in separate source modules if the modules contain SQL statements. Using the same name can cause SQL internal data structures to be interpreted as duplicates. Consequently, the SQL statements in one of the functions are not included in the object file.

### *list-file*

is the file that receives the compiler listing. The default is the default output file (usually, your home terminal); *list-file* can also be a class SPOOL DEFINE name.

### *run-option*

is one or more TACL RUN command options (separated by commas) as described in the *TACL Reference Manual*.

### *object*

specifies the object file to which the TNS C compiler writes the compilation unit.

If you omit *object*, the compiler creates a file named OBJECT in your default volume and subvolume. If the compiler cannot create OBJECT (usually, because a file with this name already exists and cannot be purged), the compiler creates a file, ZZBI $nnnn$ , in your default volume and subvolume (where  $nnnn$  is a 4-digit number determined by the system).

### *compiler-option*

is a TNS C compiler pragma or preprocessor symbol.

For single-module programs that run in the Guardian environment, specify the RUNNABLE pragma to generate an executable program object file.

To direct the TNS C compiler to generate an object file that runs in the OSS environment, specify the SYSTYPE OSS pragma.

For more information about TNS C compiler pragmas or preprocessor symbols, see the *C/C++ Programmer's Guide for NonStop Systems*.

## Running the TNS/R NMC and TNS/E CCOMP Compiler in the Guardian Environment

To run the TNS/R NMC compiler in the Guardian environment, use the NMC command. To run the TNS/E CCOMP compiler (available only on H-Series RVUs) in the Guardian environment, use the CCOMP command:

```
[ RUN ] {NMC|CCOMP} / IN source [, OUT list-file ] [ , run-
option] [object ] [ ; compiler-option [ , compiler-option
]... ]
compiler-option: one of
    pragma
    define identifier [ constant ]
    undefine identifier
```

IN *source*

specifies the primary source file of the module. This file must be a valid Guardian disk file of type 101 (edit) or type 180 (C-format). Interactive input from a terminal or a process is not accepted.

OUT *list-file*

specifies the file to which the native C compiler writes the compiler listing. When specified, *listing* is usually a spooler location. If you omit the OUT option, the compiler writes the listing to your current default output file. If the file already exists, the compiler tries to delete the file and then continue.

*run-option*

is one or more TACL RUN command options (separated by commas), as described in the *TACL Reference Manual*.

*object*

specifies the file to which the native C compiler writes the object code for the source text. If you do not specify an object file, the compiler writes the object code to the file OBJECT in your current default volume and subvolume. If the file OBJECT cannot be created, the compiler writes the object code to the file ZZBI $nnnn$  (where  $nnnn$  is a unique four-digit number) in your current default volume and subvolume.

*compiler-option*

modifies the compiler operation by specifying a compiler pragma or defining a preprocessor symbol as follows:

*pragma*

is any valid compiler pragma.

NonStop SQL/MP supports Tandem floating-point format but not IEEE floating-point format. The floating-point format for TNS/R native compilation is Tandem by default. However, for TNS/E native compilation, the floating-point format is IEEE by default. Follow these guidelines when compiling C programs that contain embedded SQL/MP statements:

- For TNS/R native compilation, do not specify the IEEE\_FLOAT compiler pragma. The floating-point format must be the default TANDEM\_FLOAT.
- For TNS/E native compilation, specify the TANDEM\_FLOAT compiler pragma to override the default IEEE\_FLOAT.

For more information, see “Compiling and Linking Floating-Point Programs” in the C/C++ Programmer’s Guide.

*define identifier [ constant ]*

defines *identifier* as a preprocessor symbol. If *identifier* is followed by a constant, it is defined as an object-like macro that expands to the given value. The *define* option is equivalent to using the `#define` preprocessor directive in source text.

*undefine identifier*

deletes *identifier* as a preprocessor symbol. The *undefine* option is equivalent to using the `#undef` preprocessor directive in source text.

## Binding SQL Program Files in the Guardian Environment

The Binder program is a tool you can use to read, link, modify, and build executable object files in the TNS environment. You can bind C, COBOL, Pascal, and TAL object files, including SQL program files.

To bind object files in the TNS/R environment, use the native link utilities (`nld` and `ld`), which are described in the *nld Manual* and the *ld Manual*. To bind object files in the TNS/E environment, use the native link utilities (`eld` and `eNOFT`) which are described in the *eld Manual* and *eNOFT Manual*.

Follow these guidelines when you bind or link SQL program files:

- Handle SQL program files like other object files.
- Bind object files after they are compiled by the TNS C compiler. (You can bind object files after running the SQL compiler. However, the binding operation invalidates the resulting target file, and you must then explicitly recompile the program file to validate it.)
- SQL compile only the final bound object. In other words, do not separately SQL compile each object of a multiple-module program.

- Do not bind object files with functions that have the same name and contain embedded SQL statements. The SQL compiler uses the function name as the run-time data unit (RTDU) name. Therefore, when the SQL statement runs, functions with the same name generate ambiguous references that can cause run-time SQL errors.

How you use the Binder program differs, depending on whether you are binding a single-module program or a multiple-module program.

For a single-module program, the C compilers automatically invoke the Binder program and generate an executable object file if you specify the RUNNABLE pragma. You specify the RUNNABLE pragma in the source code file or as a compiler option in the TACL RUN command line when you compile your program:

```
C / IN csrc, OUT $s.#clist, NOWAIT / cobj; RUNNABLE
```

If you do not specify the RUNNABLE pragma when you compile a program, you must explicitly use the Binder program or linker to set this attribute in the object file.

For a multiple-module program, use the Binder program or linker to combine the object code from each module into a single executable object file.

To run the Binder program, enter the BIND command at the TACL prompt. The Binder program displays its banner and prompt, an at sign (@).

In the next example, the Binder commands combine the `cobj1` and `cobj2` files into an executable object file named `progfile` and set the RUNNABLE attribute for `progfile`. The SELECT LIST \* OFF command improves performance by turning all listings off.

```
@ADD * FROM cobj1
@ADD * FROM cobj2
@SELECT RUNNABLE OBJECT ON
@SELECT LIST * OFF
@BUILD progfile
```

---

△ **Caution.** The Binder STRIP command without the SYMBOLS or AXCEL option removes the Binder table from an SQL program file. Without the Binder table, the SQL compiler cannot compile the program file, and the SQL executor cannot run it.

---

For more information about the Binder program, see the *Binder Manual*.

## Running the SQL Compiler in the Guardian Environment

The SQL compiler (SQLCOMP) compiles SQL source statements in a program file, generates SQL object code for each statement, determines an optimized execution plan for each SQL statement against the database, and stores the code and plan in the SQL object program. Optionally, you can invoke the EXPLAIN utility during SQL



compilation to generate a report on the execution plans for DML statements and DEFINES used by the program.

---

**Note.** The Accelerator and the Object Code Accelerator (OCA) invalidate an SQL program file. If you plan to run the Accelerator or OCA on a program file, run it before you explicitly SQL compile the program.

---

## Required Access Authority

To run the SQL compiler, 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, or index that the SQL program file uses is registered

## SQL Compiler Functions

- Resolves and expands SQL object names, including DEFINES, using the current defaults and the current catalog, and then stores the DEFINE names in the SQL program file.
- Performs type checking for C and SQL data types.
- Expands views.
- Checks references in catalogs for SQL object names to verify their existence and to read their descriptions, then evaluates the object type and characteristics for each reference.
- Determines an optimized execution plan by analyzing the DML 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 dependencies for tables, views, and indexes in the USAGES table for each table, view, or index that is accessed.
- Generates a listing of the SQL statements in the program file, including any warning or error messages that occurred.
- Sets the SQL SENSITIVE and SQL VALID flags in the program file label if the compilation is successful.

## Entering the SQLCOMP Command

To run the SQL compiler in the Guardian environment, enter the SQLCOMP command at the TACL prompt or from an OBEY command file by using this syntax. (For information about running the SQL compiler using the `c89` utility in the OSS environment, see [Developing a C Program in the OSS Environment](#) on page 6-28.)

```
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 or a system library. The object file can be generated by the C compiler, Binder program, Accelerator, OCA, or 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 your current default values.

### *list-file*

identifies the destination where 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) as follows:

`[\system.] external-file`

`\system` is an optional system name. `external-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 produce 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*.

CATALOG *catalog-name*

specifies the name of the catalog where the program is to be registered. *catalog-name* is a subvolume name. If you partially qualify the catalog name, the system expands the name by using your current default values.

You can also specify a class CATALOG DEFINE for *catalog-name*.

The catalog, object file, and SQL compiler must reside on the same system.

If the program was previously SQL compiled and recorded in a different catalog, *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 TACL DEFINES used to interpret DEFINE names in the SQL statements in the program file.

CURRENTDEFINES (the default) selects the current set of TACL DEFINES for compiling the program.

STOREDDEFINES selects the set of TACL DEFINES used for SQL tables and views that were stored in the program file the last time it was SQL compiled. (SQLCOMP does not store the settings for the `=_DEFAULTS` DEFINE in the program file.) STOREDDEFINES applies only to programs that have been SQL compiled.

```
[ EXPLAIN                                     ]
[      [ PLAN ]                             ]
[      [ DEFINES [ file-name ] [, OBEYFORM ] ] ]
[                                           ]
[ NOEXPLAIN                                ]
```

controls whether the SQL compiler invokes the EXPLAIN utility.

EXPLAIN PLAN

invokes the EXPLAIN utility to generate an EXPLAIN listing of the optimized execution plans determined by the SQL compiler for the DML statements in the program. EXPLAIN PLAN is the default EXPLAIN option.

EXPLAIN DEFINES [ *file-name* ] [, OBEYFORM ]

invokes the EXPLAIN utility to generate a listing of the TACL DEFINES that the SQL compiler uses to compile the SQL statements. (The SQL compiler uses these DEFINES to recompile the program if you specify the STOREDDEFINES option.)

*file-name* is an optional file where the SQL compiler writes the DEFINE listing. *file-name* is an *external-file* as described for the OUT *list-file* parameter.

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 later 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 (the default) suppresses the EXPLAIN utility.

FORCE | NOFORCE

controls how syntax 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 run at run time. Use the FORCE option to debug a program if you do not need to run the SQL statements that generate errors.

NOFORCE (the default) directs the SQL compiler to produce the SQL object code only if there are no syntax errors.

OBJECT | NOOBJECT

controls whether the SQL compiler produces an SQL program file.

OBJECT (the default) directs the compiler to generate an SQL object code, depending on whether errors occur and whether the FORCE or NOFORCE option is in effect.

NOOBJECT directs the compiler to perform checking functions and to generate an EXPLAIN listing if you have also specified the EXPLAIN option but to not produce SQL object code.

#### RECOMPILE | NORECOMPILE

specifies whether the program should be automatically recompiled, if necessary, during program execution.

RECOMPILE (the default) directs the SQL executor to automatically recompile a program whenever any of these conditions occur:

- The program file is SQL invalid at SQL load time.
- The DEFINEs 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.

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 compiler 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 recompile an entire invalid program or only those SQL statements that require recompilation and are actually run. If you specify NORECOMPILE, this option is ignored.

RECOMPILEONDEMAND directs the SQL executor to recompile only those statements in the invalid program that actually run. Automatic recompilation occurs the first time an individual SQL statement is run.

RECOMPILEALL (the default) directs the SQL executor to automatically recompile the entire program if it is invalid. Automatic recompilation occurs at SQL load time.

#### REGISTERONLY { ON | OFF }

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 SQL/MP software version 310 (or later).

REGISTERONLY 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 after it was SQL compiled, the operation fails with SQL error 2103.

REGISTERONLY OFF (the default) directs the SQL compiler to explicitly compile the program and perform all SQL compiler functions.

`NOREGISTER { ON | OFF }`

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).

NOREGISTER 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 after it was SQL compiled, the operation fails with SQL error 2103.

NOREGISTER OFF (the default) directs the SQL compiler to explicitly compile the program and perform all specified compiler functions, including registering the program in the catalog.

`CHECK { INVALID PROGRAM | INVALID PLANS | INOPERABLE PLANS }`

determines the behavior of the SQL executor when it runs 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). A version 310 SQL compiler sets the program's file version (PFV) to 310. If you specify the CHECK INVALID PLANS or CHECK INOPERABLE PLANS option (which stores similarity information in the program file), the SQL compiler also sets the program's catalog version (PCV) to 310 (or later). To support the CHECK INVALID PLANS or CHECK INOPERABLE PLANS option, 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.

#### CHECK INVALID PROGRAM

(the default) 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 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 check.
- The statement references a `DEFINE` at SQL load time that has changed since the last explicit SQL compilation.

The SQL executor uses the execution plans from the program file for other SQL statements that are valid.

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).

#### CHECK 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 check.
- 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 runs the statement without automatically recompiling it.

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 runs 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 { PROGRAM [ STORE SIMILARITY INFO ] }
        { INVALID PLANS
        { INOPERABLE PLANS

```

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). A version 310 SQL compiler sets the PFV to 310. 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 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).

#### COMPILE PROGRAM

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. COMPILE PROGRAM is the default.

#### COMPILE 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. (The SQL compiler also generates empty sections for CONTROL directives and DDL statements.)

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 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.

#### `COMPILE 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.

---

**Note.** Safeguard protection for a program object file might be lost after SQL compilation in certain cases. For example, if the `PROGID` bit is set for the file or if the original program cannot be modified because it is held open, you must explicitly restore Safeguard protection after SQL compilation.

---

## Using Current Statistics

For the SQL compiler to generate the best execution plan, it must have the current statistics for referenced tables. NonStop SQL/MP does not automatically update these statistics. A program must run the `UPDATE STATISTICS` statement to generate current statistics in a catalog.

To run the `UPDATE STATISTICS` statement, a program's `PAID` must meet this criteria:

- Have read access to the table and 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)

In this example, 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 `ODETAIL` table.

```
EXEC SQL UPDATE ALL STATISTICS FOR TABLE =orders;
EXEC SQL UPDATE STATISTICS FOR TABLE =odetail;
```

For more information, see the UPDATE STATISTICS statement in the *SQL/MP Reference Manual*.

## Using a PARAM Command

You can use a TACL PARAM command to specify the BINSERV program and the swap-file subvolume the SQL compiler uses for explicit SQL compilations. Use the following syntax to enter a PARAM command before you run the SQL compiler. To see the parameters currently defined, enter a PARAM command without any parameter name and value pairs. A PARAM command does not apply to automatic SQL recompilation or dynamic SQL compilation.

```
PARAM [ param-name param-value [, param-name param-value
] ...]
```

*param-name param-value*

are parameter name and value pairs. These pairs apply to the SQL compiler:

```
BINSERV guardian-name
SWAPVOL subvol
```

BINSERV *guardian-name*

specifies the BINSERV program file the SQL compiler uses during compilation. These criteria apply to *guardian-name*:

- If *guardian-name* designates a system other than the system on which the SQL compiler is running, the SQL compiler ignores the BINSERV parameter.
- If *guardian-name* does not include a volume or subvolume name, the SQL compiler uses current default values.

The default value for *guardian-name* is the BINSERV program file on the same subvolume as the SQL compiler.

```
SWAPVOL subvol
```

is a subvolume for temporary (swap) files. If you do not specify a SWAPVOL subvolume, the SQL compiler uses the default subvolume for temporary files.

This PARAM command specifies the `$sql.utils.binserv` program file and the `$sql.scratch` subvolume for the subsequent SQLCOMP process:

```
PARAM BINSERV $sql.utils.binserv, SWAPVOL $sql.scratch
...
SQLCOMP /IN cobj,OUT $s.#clst,NOWAIT/ CATALOG $sql.sqlcat
```

For more information about the PARAM command, see the *TACL 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.

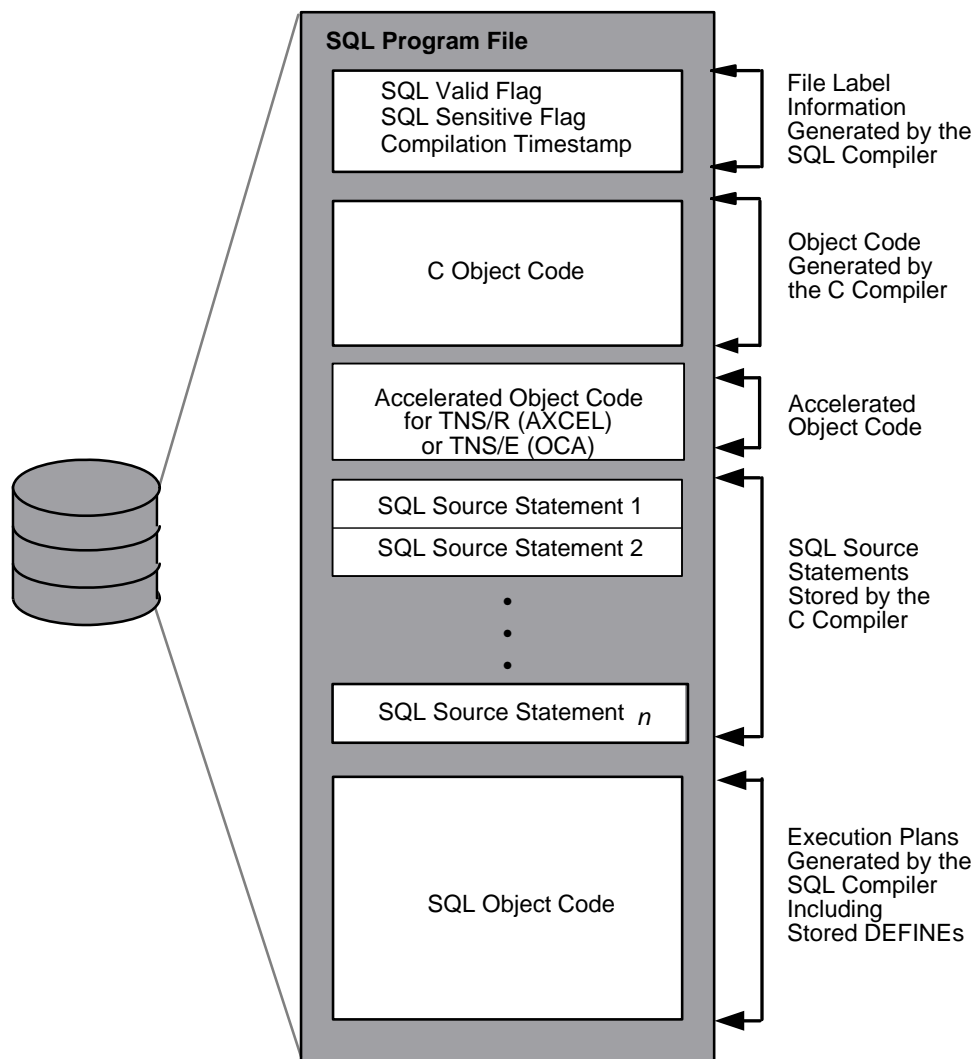
At run time, the uncompiled statement causes an error in these situations:

- Insufficient information. The SQL compiler does not have enough information to determine the validity of a statement. For example, an unavailable 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 C object file, a file generated by the Binder program, a file generated by the Accelerator, the OCA, or a file previously compiled by the SQL compiler. [Figure 6-4](#) shows the format of an SQL program file.

**Figure 6-4. SQL/MP Program File Format**



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 required.

[Example 6-1](#) shows a sample SQL compiler listing.

---

### Example 6-1. Sample SQL Compiler Listing (page 1 of 2)

```
SQL Compiler - T9095D42 - (03JUN96)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1987-1996
DATE - TIME : 10/04/96 - 16:38:23
Options : NOFORCE, OBJECT, CURRENTDEFINES, RECOMPILE, RECOMPILEALL,
REGISTERON
          NOEXPLAIN, COMPILE PROGRAM

SQL - PROGRAM FILE      = \NEWYORK.$DISK1.SQLPROG.SQLC
SQL - PROGRAM CATALOG   = \NEWYORK.$DATA1.INVENT
SQL - DEFAULT CATALOG   = \NEWYORK.$DATA1.INVENT

*****

SQL - Source File = \NEWYORK.$DISK1.SQLPROG.SQLC

SQL - SLT Index      = 0,   Run-Unit   = __SQLRTDU_00752097902906430808

49              DECLARE GET_SUPPLIER_CURSOR CURSOR FOR
50              SELECT SUPPNUM,
51              SUPPNAME,
52              STREET,
53              CITY,
54              STATE,
55              POSTCODE
56              FROM =SUPPLIER
57              WHERE SUPPNUM = :supplier_of_parts
58              REPEATABLE ACCESS
*** Statistics: Estimated cost: 1

SQL - SLT Index      = 0,   Run-Unit   = abort_transaction
184              ROLLBACK WORK

SQL - SLT Index      = 0,   Run-Unit   = begin
99              BEGIN WORK

SQL - SLT Index      = 0,   Run-Unit   = commit_transaction
163              COMMIT WORK

SQL - SLT Index      = 0,   Run-Unit   = do_add_to_partloc
```

---

---

**Example 6-1. Sample SQL Compiler Listing (page 2 of 2)**

```

133                                INSERT INTO =PARTLOC
134                                VALUES (:partloc_rec.loc_code,
135                                        :partloc_rec.partnum,
136                                        :partloc_rec.qty_on_hand)
*** Statistics: Estimated cost: 1

SQL - SLT Index    = 0,    Run-Unit  = do_add_to_parts

149                                INSERT INTO =PARTS
150                                VALUES (:parts_rec.partnum,
151                                        :parts_rec.partdesc,
152                                        SETSCALE
(:parts_rec.price,2)
153                                :parts_rec.qty_available)
*** Statistics: Estimated cost: 2
BINDER - OBJECT FILE BINDER - T9621D30 - (17JUL95)    SYSTEM \NEWYORK
Copyright Tandem Computers Incorporated 1982-1995

Object file \NEWYORK.$DISK1.SQLPROG.SQLC
TIMESTAMP 1996-10-04 16:34:56

PAGE    1                                10/04/96 - 16:38:23

        0 Binder Warnings
        0 Binder Errors

PAGE    2                                10/04/96 - 16:38:23

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             = 00:00:00.264
SQL -   Elapsed Time             = 00:00:18.096
SQL -   Program file is \NEWYORK.$DISK1.SQLPROG.SQLC
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](#) on page 6-27
- [EXPLAIN DEFINES Report](#) on page 6-27

## EXPLAIN PLAN Report

The EXPLAIN PLAN report, which applies only to DML statements, indicates 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, including one for the statement itself and one for each subquery. 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 a program file, `sqlprog`, 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.#sqlist / CATALOG $disk2.sales,
        EXPLAIN PLAN
```

## EXPLAIN DEFINES Report

The EXPLAIN DEFINES report indicates the mapping of DEFINE names used in SQL statements with this information:

- Each DEFINE name and its associated Guardian name used for SQL tables and views
- The default volume and default catalog used by the SQLCOMP process (which it gets from the current `=_DEFAULTS DEFINE`)

The EXPLAIN utility can generate EXPLAIN DEFINES reports in these formats:

OBEY command file format    EXPLAIN generates the ADD DEFINE commands that add DEFINES. You can then use a TACL OBEY command to run 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 report. In an actual report, each *subvol-name*, *guardian-name*, and *define-name* would be replaced by the actual name.

```
ALTER DEFINE =_DEFAULTS, VOLUME  subvol-name
ALTER DEFINE =_DEFAULTS, CATALOG subvol-name
```

```
ADD DEFINE  define-name, FILE guardian-name
ADD DEFINE  define-name, FILE guardian-name
...
```

When you issue an OBEY command to run the file shown in the next example, 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 compiler also writes the DEFINES in OBEY command file format to the file `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 program 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, see the *SQL/MP Query Guide*.

## Developing a C Program in the OSS Environment

Version 315 (or later) SQL/MP software supports the development of C programs in the OSS environment. You can code a C program that contains embedded SQL statements with a text editor such as `vi` or `ed` and then use the `c89` utility to invoke the C and SQL compilation tools. (C++ does not support embedded SQL statements.)

- TNS/R native C programs require version 2 (or later) SQL/MP software
- TNS/E native C programs require version 350 SQL/MP software

You can also develop a C program in the Guardian environment that runs in the OSS environment by specifying the `SYSTYPE OSS` pragma when you compile the program. For more information, see [Developing a C Program in the Guardian Environment](#) on page 6-5.



For information on how to compile and link programs in the OSS environment, see the `c89` (1) reference pages online or in the *Open Systems Services Shell and Utilities Reference Manual*. If you are migrating a program from the TNS environment to the TNS/R or the TNS/E environment, see the *TNS/R Native Application Migration Guide* and the *H-Series Application Migration Guide*.

## Using TACL DEFINES in the OSS Environment

In the OSS environment, a C program can contain class MAP and class CATALOG DEFINES. Use these OSS utilities to create and manipulate these DEFINES:

OSS Utility	Description
<code>add_define</code>	Creates a new class MAP, CATALOG, SPOOL, SORT, SUBSORT, SEARCH, or TAPE DEFINE
<code>del_define</code>	Deletes one or more DEFINES
<code>info_define</code>	Displays the attributes and values of existing DEFINES
<code>set_define</code>	Sets the values for one or more DEFINE attributes in the current working attribute set
<code>show_define</code>	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 attributes and the associated values. For a detailed description, including the syntax of these utilities, see the *Open System Services Shell and Utilities Reference Manual* or the appropriate reference pages.

Considerations for using TACL DEFINES in the OSS environment:

- The `add_define` utility implicitly sets the DEFMODE attribute to ON before it creates the new DEFINE.
- Before you run the C compiler using the `c89` 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 `c89` utility, add the DEFINE before you issue the `c89` 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 `add_define` commands create a class MAP DEFINE and a class CATALOG DEFINE:
 

```
add_define =emptab class=map file=\\ny.\\$dsk2.fy94.empfile
add_define =sqlcat class=catalog subvol=\\$sql.sqlcat
```
- System names or the names of volumes where OSS objects reside must be seven characters or fewer.

- 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 the c89 Utility in the OSS Environment

The `c89` utility is the OSS driver for the C and C++ compilation systems. You use the `c89` utility to run the C and C++ compiler or `ld`, the TNS/E native linker (`eld`), Accelerator, OCA, and SQL compiler.

With D40 and later product versions, `c89` utilities are available in these ways:

- The native `c89` utility resides in the `/usr/bin` directory. For information about the native `c89` utility, see the *Open System Services Shell and Utilities Reference Manual* or the `c89(1)` reference pages.
- The TNS `c89` utility has been moved from the `/bin` directory to the `/nonnative/bin` directory. The documentation for the TNS `c89` utility is available only as reference pages. To view the TNS `c89` reference pages, enter:

```
man -M /nonnative/usr/share/man c89
```

## Running the OSS C Compiler

Run the OSS C compiler using the `c89` utility to compile the source file (or files) that contain the embedded SQL statements. The C compiler generates an object file that contains C object code and SQL source statements. This `c89` command invokes the C compiler to compile the `pgm1` source file into the `pgm1.o` object file:

```
c89 -c pgm1
```

NonStop SQL/MP supports Tandem floating-point format but not IEEE floating-point format. The floating-point format for TNS/R native compilation is Tandem by default. However, for TNS/E native compilation, the floating-point format is IEEE by default. Follow these guidelines when compiling C programs that contain embedded SQL/MP statements:

- For TNS/R native compilation, do not specify the `-WIEEE_float` flag. The floating-point format must be the default `TANDEM_FLOAT`.
- For TNS/E native compilation, specify the `-WTandem_float` flag to override the default `IEEE_FLOAT`:

```
c89 -WTandem_float pgm1
```

For more information, see “Compiling and Linking Floating-Point Programs” in the *C/C++ Programmer’s Guide for NonStop Systems*.

The native `c89` utility resides in the `/bin/compilers` directory. The TNS C compiler has been moved from the `/bin/compilers` directory to the `/nonnative/bin/compilers` directory, and the TNS C run-time library has been moved from `/usr/lib/libc.a` to `/nonnative/usr/libc.a`.

For more information about the C compiler, see the *C/C++ Programmer's Guide for NonStop Systems*.

## Running the Binder Program, `nld`, `ld`, or `eld` Utility

In the TNS environment, use the TNS `c89` utility to run the Binder (BIND) program to combine multiple object files into one target object file. This `c89` command invokes the Binder program to bind the `pgm1o` and `pgm2o` object files into the `sqlprog` file and to set the HIGHPIN attribute to ON:

```
c89 -o sqlprog -Wbind="set highpin on" pgm1o pgm2o
```

In the TNS/R environment, use the native `c89` utility to run the native link utility (`nld` or `ld`) to link multiple object files and produce an executable object file. For information about the `nld` or `ld` utility, see the *nld Manual* or *ld Manual*.

In the TNS/E environment, use the native `c89` utility to run the TNS/E native linker utility (`eld`) to link multiple object files and produce an executable object file. Use `eNOFT` to read, display linkfiles, loadfiles, and import libraries created by the TNS/E C compiler. For information about the `eld` utility and `eNOFT`, see the *eld Manual* and the *eNOFT Manual*.

Do not bind object files with functions that have the same name and contain embedded SQL statements. The SQL compiler uses the function name as the RTDU name. Therefore, when the SQL statement runs, functions with the same name generate ambiguous references, which can generate SQL run-time errors.

## Running the Accelerator for Cross-Platforms

TNS object files can be accelerated to take advantage of features on the RISC architecture for TNS/R or on the Intel Itanium architecture for TNS/E. In most cases, accelerated TNS objects have significant performance benefits when optimized for target TNS/R and TNS/E platforms.

For a program compiled in the TNS environment, you can run the Accelerator (AXCEL) to improve the program's performance when it runs in the TNS/R environment. You run the OCA to accelerate TNS object files on the TNS/E platform for execution on the TNS/E platform.

This TNS `c89` command invokes the Accelerator to accelerate the `sqlprog` file without initiating the binding process:

```
c89 -Wnobind -Waxcel sqlprog
```

Because the Accelerator and the OCA invalidate SQL program files, run the accelerators before you explicitly SQL compile a program to avoid having to recompile.

You can also accelerate a program file by specifying the `-O` flag when you run the C compiler. For more information about the TNS/R accelerator, see the *Accelerator Manual*. For more information about the TNS/E accelerator, see the *Object Code Accelerator (OCA) Manual*.

## Running the SQL Compiler

Use the `c89` utility to run the SQL compiler (`sqlcomp`) to explicitly compile embedded SQL statements in the C object file. The SQL compiler validates the program file for execution and registers the program in the PROGRAMS and USAGES catalog tables using its Guardian ZYQ name.

With the D40 and later D-series product versions, the native `c89` and TNS `c89` utilities use different flags to run the SQL compiler, as described next.

### Using the Native c89 Utility

To compile a C program with embedded SQL statements using the native `c89` utility, include the `-Wsql` flag to specify the SQL pragma and the `-Wsqlcomp` flag to run the SQL compiler:

```
c89 -Wsql=sqlmap,release2
    -Wsqlcomp="catalog \ $sql.sqlcat,recompileondemand" sqlprog.c
```

For more information about the native `c89` utility, see the *Open System Services Shell and Utilities Reference Manual* or the `c89(1)` reference pages.

### Using the TNS c89 Utility

With the D40 and later D-series product versions, the TNS `c89` utility also uses different flags from earlier versions of the `c89` utility. For example, to compile an SQL program file without invoking the Binder program, specify the `-Wsqlcomp` flag to run the SQL compiler by using the catalog `$sql.sqlcat`:

```
c89 -Wnbind -Wsql="catalog \ $sql.sqlcat" sqlprog.c
```

For more information about the TNS `c89` utility, see the reference pages by entering this command:

```
man -M /nonnative/usr/share/man/ c89.1
```

## Considerations for Running the SQL Compiler

Other considerations for using the `c89` utility to invoke the SQL compiler:

- You must precede a dollar sign (\$) in a catalog name with the OSS shell escape character (\). For example:
- ```
c89 -Wsqlcomp="catalog \ $sql00.sqlcat" sqlprog.c
```
- Although the SQL compiler accepts an OSS path name for an input object file, you must use Guardian names or DEFINES to refer to SQL objects and catalogs in embedded SQL statements within the program.
  - The input object file to the SQL compiler can be either an OSS file (code 180) or a Guardian file (code 101). However, if you specify a Guardian file as input, you must use its OSS path name format:

```
/G/volume/subvolume/file-id
```

- The input object file also determines the environment where the resulting SQL program file resides after explicit SQL compilation. If the program file resides in the Guardian environment, the SQL compiler uses the OSS path name format. The OSSFILE column in the PROGRAMS table indicates the environment where the file resides (Y=OSS, N=Guardian).
- To specify a program's target execution environment (that is, the environment where the program runs), set the `c89 -Wsystype` flag as follows:

```
oss          OSS environment (the default)
guardian     Guardian environment
```

- The SQL compiler directs all error information and listings to the `c89` process and does not directly use the home terminal.
- To SQL compile a C program using the `c89` utility, use SQL compiler version 315 (or later). After SQL compilation, an OSS program file has a PFV and a PCV of 315 (or later).
- To use the EXPLAIN utility, you must also specify the `-Wverbose` flag:

```
c89 -Wsql -Wverbose -Wsqlcomp="explain plan" -Wnolink
sqlprog.c
```

- To compile embedded SQL/MP for TNS/E, the value specified for `sqlhost` must be an H-series TNS/E system.
- The `c89` utility on UNIX workstations does not support the compilation of programs that contain embedded SQL statements.

For more information about the `c89` utility, see the *C/C++ Programmer's Guide for NonStop Systems*.

## **-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: `c89`, and `c99`.

The syntax is:

```
-Wsqlconnect = mode
```

Where `mode` is:

|                           |                                                                                                                                                                                                                                          |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>legacy</code>       | Directs the compiler to connect using the legacy (unencrypted) mode.                                                                                                                                                                     |
| <code>secure_quiet</code> | 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.                                       |
| <code>secure_warn</code>  | 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. |
| <code>secure_err</code>   | 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

The usage considerations for `Wsqlconnect` are:

- This option requires both the `-Wsqlhost` and `-Wsqluser` options to be specified. If an invalid value is specified, an error is returned.
- If the value of `-Wtarget` is `tns/r` or `mips`, a secure connection is not available.
  - If the `-Wsqlconnect= secure_err` is specified, an error is returned.
  - If the `-Wsqlconnect= secure_warn` is specified, a warning 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.

## Developing a C Program in a PC Host Environment

You can compile SQL/MP applications on the PC by using either the HP Enterprise Toolkit—NonStop Edition (ETK) or the command-line cross compiler (`c89`) directly from the command line in Windows. You must connect to an HP NonStop operating system host (TNS/R or TNS/E) for SQL compile time operations and to run an application. The resulting object files can be executed on NonStop TNS/R and TNS/E native systems.

ETK is a GUI-based extension package to the Visual Studio.NET product. You can use ETK to edit, compile, build, and deploy applications written in C and COBOL with embedded SQL/MP. You do not have to install Visual Studio.NET or ETK to use the command-line interface. For more information, see the online help in ETK or the file “Using Command-Line Cross Compilers” installed with the ETK compiler package. For command-line help, enter: `c89 -Whelp`.

## Using CONTROL Directives

You can use CONTROL directives with either static or dynamic DML statements. However, CONTROL directives do not affect DDL statements. The CONTROL directives and their functions are:

### CONTROL EXECUTOR

allows or prohibits parallel evaluation 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

For the syntax of each CONTROL directive, see the *SQL/MP Reference Manual*.

## Static SQL Statements

Follow these guidelines when you use a static CONTROL directive with static SQL statements in a C program:

- The scope of a static CONTROL directive is the program's current RTDU. An SQL map shows each RTDU. To generate an SQL map in the C compiler listing, specify the SQLMAP option in the SQL directive.

A static CONTROL directive affects all subsequent static SQL statements that follow in listing order (regardless of execution order) as follows:

- Global Scope—In the global scope of a C program (that is, outside of any functions), a static CONTROL directive affects only SQL statements that follow in listing order and are not within a function. It does not affect SQL statements within a function.
- Functions—Within a C function (which is a unique RTDU), a CONTROL directive affects subsequent static SQL statements in the function in listing order until the end of the function or until another CONTROL directive resets the directive. A static CONTROL directive outside of a function does not affect SQL statements in the function.
- A CONTROL directive coded within flow-control statements (for example, IF and ELSE) applies to static 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 (and in the same RTDU as the DECLARE CURSOR statement).
- A dynamic CONTROL directive does not affect static SQL statements in the program except as described under [Dynamic SQL Statements](#) on page 6-36.



In this example, the **CONTROL EXECUTOR** directive specifies parallel evaluation when the program runs the first **FETCH** statement for the cursor.

```
EXEC SQL CONTROL EXECUTOR PARALLEL EXECUTION ON;
EXEC SQL DECLARE list_customers_with_orders CURSOR FOR
    SELECT CUSTOMER.CUSTNUM, CUSTOMER.CUSTNAME
    FROM   =CUSTOMER, =ORDERS
    WHERE  CUSTOMER.CUSTNUM = ORDERS.CUSTNUM
    STABLE ACCESS;
```

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*).

```
...
/* 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;

/* Short wait... */
EXEC SQL CONTROL TABLE sales.parts TIMEOUT .1 SECOND;

EXEC SQL
    DECLARE CURSOR cursor2
    FOR SELECT partnum,partdesc,price
    FROM sales.parts
    WHERE (partnum > :min_partnum AND partnum < :max_partnum)
    ORDER BY partnum;

/* Infinite wait....*/
EXEC SQL CONTROL TABLE sales.parts TIMEOUT -1 SECOND;

EXEC SQL
    DECLARE CURSOR cursor3
    FOR SELECT partnum,partdesc,price
    FROM sales.parts
    WHERE (partnum > :min_partnum AND partnum < :max_partnum)
    ORDER BY partnum;
...
```

## 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 RTDU that contains the CONTROL TABLE directive.

---

## Using Compatible Compilation Tools

### C Compiler

The host SQL version (HSV) identifies the SQL version of the C compiler. A C program that uses version 300 (or later) SQL features must be compiled with a C compiler that has an HSV of 300 (or later). To determine the HSV of the C compiler, use one of these methods:

- Run the VPROC program for the C compiler object file. VPROC displays a line for each object file bound into the target object file. For the C compiler, check the version in the VPROC line that contains S7094, which is the SQL compiler interface (SCI) product number.
- When you run the C compiler, specify the SQLMAP option in the SQL compiler directive. The SQLMAP option directs the C compiler to include the HOSV in the map at the end of the source-file listing. For example, a version 310 C compiler listing includes this line:

```
Host Object SQL Version = 310
```

### 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 the NonStop operating system, including the SQL compiler, must have the same version.

### SQL Program Files

An SQL program file has these versions:

---

|      |                                                                                                                                                                        |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HOSV | The version of the C compiler used to compile the program. Generated by the C compiler and therefore the same as the host SQL version (HSV) of the C compiler.         |
| PFV  | The version of the SQL compiler used to compile the program. Indicates the oldest version of the SQL executor that can run the program. Generated by the SQL compiler. |
| PCV  | The oldest version of the SQL catalog in which the program can be registered. Generated by the SQL compiler.                                                           |

The HOSV and its relationship to the C compiler and SQL compiler are described next. For more information about the PFV and PCV, see the *SQL/MP Version Management Guide*.

The C compiler generates the HOSV and stores the value in the object file. If multiple object files are bound together into 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 an SQL program file, use the GET VERSION OF PROGRAM statement with the HOST OBJECT option. You can run this statement from SQLCI or in a C program. This GET VERSION OF PROGRAM statement is run 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 C program, you must include the INTO clause with a host variable. This statement returns the HOSV of SQLPROG to the host variable HV\_HOSV:

```
EXEC SQL
    GET HOST OBJECT VERSION OF PROGRAM sqlprog INTO :hv_hosv;
```

You can also run a dynamic GET VERSION OF PROGRAM statement using the PREPARE and EXECUTE statements as shown:

```
strcpy (hv_text,
        "GET HOST OBJECT VERSION OF PROGRAM SQLPROG");
EXEC SQL PREPARE dynamic_statement FROM :hv_text;
EXEC SQL EXECUTE dynamic_statement RETURNING :hv_hosv;
...
```

You cannot, however, 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 NonStop C program containing embedded SQL statements and directives in the OSS environment. The section provides details about the required access permissions, the TACL DEFINES used, and the steps to run the TACL RUN command. It further explains how to run a program at low PIN and how to determine compatibility with the SQL executor.

Topics include:

- [Required Access Authority](#)
- [Using TACL DEFINES](#) on page 7-2
- [Entering the TACL RUN Command](#) on page 7-3
- [Running a Program in the OSS Environment](#) on page 7-3
- [Running a Program at a Low PIN](#) on page 7-4
- [Determining Compatibility With the SQL Executor](#) on page 7-7

## **Required Access Authority**

To run a NonStop SQL program file, you (or the creator process, if you use a process-creation procedure such as PROCESS\_CREATE\_ or NEWPROCESS) must have the following 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 (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 for both the process access ID (PAID) and the 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 TACL DEFINES

Before running an SQL program file, you can specify TACL DEFINE, PARAM, or ASSIGN commands. For information about PARAM and ASSIGN commands, see the *TACL Reference Manual*.

You can use TACL DEFINE names in an SQL program to specify the names of SQL catalogs and objects (tables, views, indexes, and partitions). Use a class CATALOG DEFINE for a catalog and a class MAP DEFINE for an object.

You enable and disable DEFINES 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 and Guardian system procedures. You can also specify the =\_SORT\_DEFAULTS DEFINE to control sort operations.

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 run 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 using this syntax.

```
[ RUN[D] ] program-file [/ [ ,run-option ].../ [ argument ]...
```

RUN

runs the program file without invoking the Inspect debugger.

RUND

runs 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 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*.

*argument*

is an argument as described in the *C/C++ Programmer's Guide*. Separate arguments in a list using spaces, not commas.

For example, this RUN command runs the program file named `sqlprog` and specifies the NAME, OUT, and NOWAIT run options:

```
RUN sqlprog / NAME $sqlrun, OUT $s.#sqlist, NOWAIT /
```

This RUND command runs the program file named `$disk.sql.sqlprog` under the control of the Inspect debugger:

```
RUND $disk.sql.sqlprog
```

For more information about the RUN command, see the *TACL Reference Manual*.

## Running a Program in the OSS Environment

To run an SQL program file in the OSS environment, enter the program file name at the OSS shell prompt. You can also use the OSS `run` command to run a program file using HP attributes (for example, a CPU or priority for the process). For information about the `run` command, see the *Open System Services Shell and Utilities Reference Manual* or the `run (1)` reference pages.

# Running a Program at a Low PIN

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 where the process is running. For example, the operating system identifies a process in processor 4 with PIN 195 as 4,195.

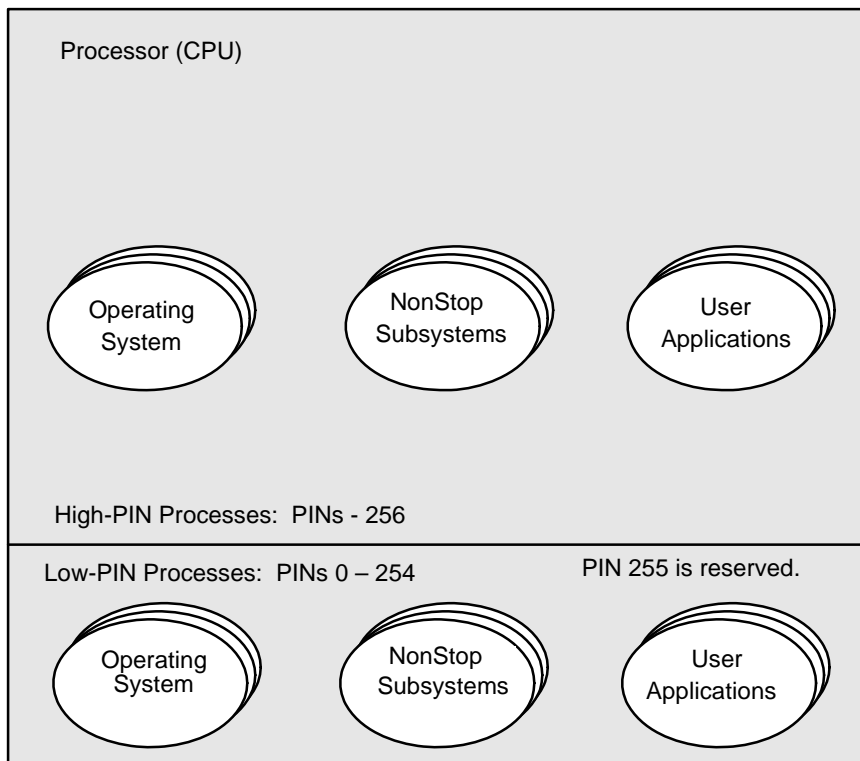
The operating system supports an architectural limit of 65,535 concurrent processes per processor. The actual number of concurrent processes depends on the available system resources (for example, virtual memory) and the values specified during system generation.

A 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.

[Figure 7-1](#) shows various processes running in a processor on an HP NonStop system.

**Figure 7-1. Processes Running on a NonStop System**



VST010.vsd



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. In some cases, however, you might need to run a program at a low PIN. For example, in a mixed network of C-series and D-series nodes, a program must run at a low PIN on a D-series node to:

- Communicate with a process on a C-series node
- Access a file or an SQL object on a C-series node

If you run an SQL program remotely on a D-series node from a C-series node, the SQL program automatically runs at a low PIN. If you run an SQL program locally on a D-series node, 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, use either of these methods:

- Before you run the SQL program, set the HIGHPIN object-file attribute to OFF in the SQL program file 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 you are starting the program programmatically, call the `PROCESS_CREATE_` procedure with bit 15 of the `create_options` parameter set to 1:

```
#include <cextdecs(PROCESS_CREATE_, ...) >
...
error := PROCESS_CREATE_(program_file:length,
                        ...,
                        create_options); /* Bit 15=1 */
```

(You can also use the `NEWPROCESS` or `NEWPROCESSNOWAIT` procedure, which always forces a new process to run at a low PIN.)

If a C program must run an SQL program programmatically at a low PIN, consider these situations:

- The C creator program was not written (or converted) to run at a high PIN.  
The SQL program runs at a low PIN by default, even if it was written (or converted) to run at a high PIN.
- The C 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.

For this program to start the SQL program at a low PIN, set the HIGHPIN object-file attribute to OFF in the SQL program file using the Binder CHANGE command as described under [Interactive Commands](#) on page 7-5.

## 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.

To force an SQL program to run at a low PIN, use either of these methods:

- In the SQL program file, set the HIGHPIN object-file attribute to OFF using the Binder CHANGE command as described under [Interactive Commands](#) on page 7-5.
- In the Pathway configuration file, set the HIGHPIN server attribute to OFF using the SET SERVER or ALTER SERVER command. (The default for the HIGHPIN server attribute is OFF.)

For more information, see the *Guardian Programmer's Guide*. For information about converting a C-series program to use D-series features, see the *Guardian Application Conversion Guide*.

# Determining Compatibility With the SQL Executor

The PFV of an SQL program indicates the oldest version of the SQL executor that can run 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 PFV, it runs the program. Otherwise, the executor returns an error.

To determine the version of the SQL executor, use the GET VERSION OF SYSTEM statement. All SQL/MP components on a system, including the executor, have the same version. You can run the GET VERSION OF SYSTEM statement from SQLCI or a C program.

For a static GET VERSION OF SYSTEM statement in a C program, include the INTO clause with a host variable:

```
EXEC SQL GET VERSION OF SYSTEM \newyork INTO :hv_sys_version;
```

In this example, the GET VERSION OF SYSTEM statement returns the version of NonStop SQL/MP installed on the \NEWYORK system to the host variable named hv\_sys\_version. 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 in a C program. To embed a static GET VERSION OF PROGRAM statement in a C program, include the INTO clause with a host variable. This statement returns the PFV of SQLPROG to the host variable hv\_pfv:

```
EXEC SQL GET FORMAT VERSION OF PROGRAM sqlprog INTO :hv_pfv;
```

You can also run a dynamic GET VERSION OF PROGRAM statement using the PREPARE and EXECUTE statements as shown in this example:

```
strcpy (hv_text, "GET FORMAT VERSION OF PROGRAM SQLPROG");
EXEC SQL PREPARE dynamic_statement FROM :hv_text;
EXEC SQL EXECUTE dynamic_statement RETURNING :hv_pfv;
```

You cannot, however, use a GET VERSION OF PROGRAM statement with the EXECUTE IMMEDIATE statement.

For the syntax of the GET VERSION statements, see the *SQL/MP Reference Manual*.



# Program Invalidation and Automatic SQL Recompilation

## Program Invalidation

A NonStop 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 SENSITIVE flag in the program's file label indicates whether the file is an SQL program that has been successfully SQL compiled (although the program 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 run.

## 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 using the FUP or SQLCI DUP command, the original file is unaffected, but the new file is invalid.
- Binding a program file. If you explicitly bind a program file using the Binder program, the original file is unaffected, but the resulting target file is invalid.
- Restoring a program file. If you restore a program file 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 (for TNS/R systems only), 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 Automatic Recompilations](#) on page 8-9:

- Adding an index to a table, including an underlying table of a protection or shorthand view, 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, 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, changing the AUDIT attribute can cause automatic SQL recompilation, if it is allowed in these cases:

- 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 run 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 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](#)
- Adding an index to a table 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 reexecute 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 runs 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 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 runs 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 run.

## 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 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 runs.)

## 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 runs 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.

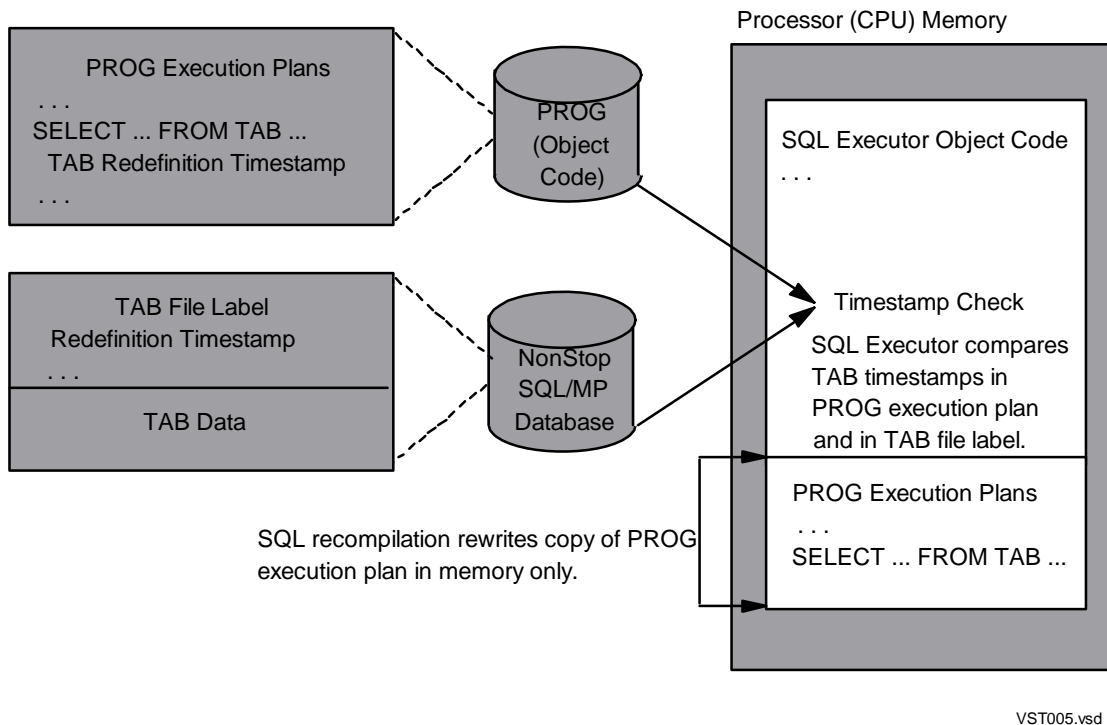
After opening a table, the SQL executor usually leaves a table open until the program stops running. However, a subsequent DDL or utility operation performed on the table (or a dependent object such as an index or collation) causes the table to be closed and its redefinition timestamp to be updated. If the SQL statement that refers to the table runs again, the SQL executor reopens the table and then performs the timestamp check to force 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, NonStop 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 runs, 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 changes only 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 run 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 automatic SQL recompilation is successful, the SQL statement runs. However, if recompilation fails, the SQL executor returns compilation errors or warnings as follows:

- 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 runs 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 runs other SQL statements using their existing plans. Executing the similarity check for an SQL statement eliminates unnecessary recompilations and is much faster than recompiling the statement.

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 using the CHECK INOPERABLE PLANS option.
2. Enable the similarity check using DDL statements for each table or protection view referenced in the program. (NonStop 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 fails the similarity check, and the SQL statement containing the query must be automatically recompiled before it can run (if NORECOMPILE is not specified). To use the similarity check in this query, you must disable parallel execution 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. (NonStop 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 is version 310 (or later). All SQL/MP components, including the executor, catalog manager, and compiler, must be version 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 following listed differences. 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—However, if 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.

## 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 these 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 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 different number of columns must observe these restrictions (in addition to 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.

Thus, 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 different number of columns as shown in these statements:

| Statement                                                                                                     | Similarity Check Results        |
|---------------------------------------------------------------------------------------------------------------|---------------------------------|
| <code>SELECT * FROM table1</code>                                                                             | TABLE1 = Fail                   |
| <code>SELECT DISTINCT * FROM table1</code>                                                                    | TABLE1 = Fail                   |
| <code>SELECT COUNT (*) FROM table1</code>                                                                     | TABLE1 = Pass                   |
| <code>SELECT columna FROM table1<br/>WHERE columnb relation-operator<br/>(SELECT COUNT(*) FROM table2)</code> | TABLE1 = Pass,<br>TABLE2 = Pass |

| Statement                                                                                             | Similarity Check Results        |
|-------------------------------------------------------------------------------------------------------|---------------------------------|
| SELECT <i>columna</i> FROM <i>table1</i><br>WHERE EXISTS<br>(SELECT [DISTINCT] * FROM <i>table2</i> ) | TABLE1 = Pass<br>TABLE2 = Fail  |
| INSERT INTO <i>table1</i><br>(SELECT [DISTINCT] * FROM <i>table2</i> )                                | TABLE1 = Fail<br>TABLE2 = Fail  |
| SELECT <i>table1</i> .*, <i>table2</i> . <i>x</i><br>FROM <i>table1</i> , <i>table2</i>               | TABLE1 = Fail,<br>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 when you specify the CHECK INOPERABLE PLANS option and execution-time name resolution. When the SQL executor tries to use the plan with a new set of tables, it retains the association of 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 columns in the table, as follows:

```
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 run 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 similarity check implicitly enabled. Two collations are similar only if they are equal. NonStop 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.



# Error and Status Reporting

This section describes error and status reporting after the execution of a NonStop SQL statement or directive in a C 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 9-13

## Using the INCLUDE STRUCTURES Directive

The INCLUDE STRUCTURES directive specifies the version of the SQL structures that the C compiler generates. You must specify the INCLUDE STRUCTURES directive to generate version 300 or later SQL data structures. If you omit this directive, the C 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 might 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 }...

    { SQLSA VERSION CURRENT }

    { { SQLCA | SQLSA } [ EXTERNAL ] }
```

#### ALL VERSION

specifies the same version for all three SQL structures (SQLCA, SQLSA, and SQLDA).

```
{ SQLCA | SQLSA | SQLDA } VERSION
```

specify the SQLCA, SQLSA, or SQLDA structure, respectively.

#### version

specifies the version number of the generated data structures; *version* can be 1, 2, 300, or 340 (or later). Version 330 applies only to the SQLSA structure.

#### SQLSA VERSION CURRENT

specifies that a subsequent INCLUDE SQLSA directive should generate both a version 300 and version 330 SQLSA structure. This option supports run-time SQLSA versioning, which allows a program to use an SQLSA structure that has the same version as the current SQL/MP software for the system.

The SQLSA VERSION CURRENT option has these requirements:

- It applies only to the SQLSA structure and not to the SQLCA or SQLDA structure.
- The SQL/MP software version must be 340 or later.
- You must compile your program using the NMC compiler on TNS/R systems or the C compiler with the CPPSOURCE option of the SQL pragma on TNS systems. For more information, see the *C/C++ Programmer's Guide*.
- You must include the SQLGETSYSTEMVERSION procedure declaration from the `cextdecs` header file in your program, because the option generates a call to this procedure. For example:

```
#include <cextdecs (SQLGETSYSTEMVERSION, ...)>
```

```
{ SQLCA | SQLSA } [ EXTERNAL ]
```

specifies that the structures are declared as external, making it possible to share them among modules of an object file. No space is allocated for an external SQLCA or SQLSA declaration. You must specify one occurrence of the formal declaration of SQLCA and SQLSA somewhere in the program using the INCLUDE SQLCA and INCLUDE SQLSA directive without the EXTERNAL option.

## Generating Structures With Different Versions

You can generate SQL structures that are all of the same version or structures of different versions. For example, to generate all version 300 structures in a program, specify:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 300;
```

Or, this directive generates different versions for each structure:

```
EXEC SQL INCLUDE STRUCTURES SQLCA VERSION 315
                                SQLDA VERSION 2
                                SQLSA VERSION CURRENT;
```

## Checking the Version of the C Compiler

If you try to compile a C program that uses the INCLUDE STRUCTURES directive to specify a later version of a structure than the C compiler can generate, the compiler returns SQL error 11203. To determine the version of the C compiler before you compile a program, run the VPROC program for the C 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 C compiler to include the HOSV of the C compiler in the map at the end of the source-file listing. For example, a version 310 C 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 modules of an object file 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. An object file with many modules that contain embedded SQL can consume an enormous amount of memory space for multiple structures alone.

The C external declaration generated by the `INCLUDE SQLCA EXTERNAL` directive is:

```
extern struct SQLCA_TYPE sqlca;
```

The C external declaration generated by the `INCLUDE SQLSA EXTERNAL` directive is:

```
extern struct SQLSA_TYPE sqlsa;
```

## Returning Error and Warning Information

NonStop SQL/MP provides these methods that you can use to process errors and warnings in a program:

- Checking the `sqlcode` variable
- Using the `WHenever` directive
- Checking information from the SQLCA structure

### Checking the `sqlcode` Variable

NonStop SQL/MP returns an error or warning code to `sqlcode` after the execution of each embedded SQL statement or directive as follows:

| Value | Status     |
|-------|------------|
| < 0   | Error      |
| > 0   | Warning    |
| 0     | Successful |

Each SQL/MP error or warning message has an assigned code. For these codes and their meanings, see the *SQL/MP Messages Manual*.

### Declaring the `sqlcode` Variable

Declare `sqlcode` as a type `short` variable within the scope of each embedded SQL statement. One method is to declare `sqlcode` as a global variable at the start of each C source module that contains embedded SQL statements.

```
#pragma SQL
/* Other pragmas, directives, and comments */
...

short sqlcode;

...
```

Use C conditional statements to check the `sqlcode` variable. [Example 9-1](#) on page 9-5 inserts two column values into the PARTS table and then checks `sqlcode` for any errors and warnings.



---

**Example 9-1. Checking the sqlcode Variable**

```

EXEC SQL INCLUDE STRUCTURES ALL VERSION 315;
/* Variable declarations: */

EXEC SQL BEGIN DECLARE SECTION;
    struct
    { short in_partnum;
      long in_price;
      char in_partdesc[19];
    } in_parts_rec;
EXEC SQL END DECLARE SECTION;

/* Include the SQLCA for detailed error information */
EXEC SQL INCLUDE SQLCA;

/* Include sqlcode for simple error checking */
short sqlcode;

...

void do_sql_insert(void)
{
/* Do an SQL INSERT into the parts table: */

    in_parts_rec.in_partnum = 4120;
    in_parts_rec.in_price   = 6000000;

/* IN_PRICE value is multiplied by 100 to reflect scale */

    strcpy (in_parts_rec.in_partdesc,"V8 DISK OPTION      ");
    EXEC SQL INSERT INTO sales.parts (partnum, price, partdesc)
        VALUES ( :in_parts_rec.in_partnum,
                  SETSCALE (:in_parts_rec.in_price, 2),
                  :in_parts_rec.in_partdesc);
/* The SETSCALE function represents the scale to SQL */

/* Check any for errors and warnings: */

    if (sqlcode < 0) handle_errors();

    if (sqlcode > 0 && sqlcode != 100) handle_warnings();

...
} /* End of do_sql_insert */

```

---

## Using the WHENEVER Directive

The WHENEVER directive specifies an action that a program takes depending on the results of subsequent DML, DCL, and DDL statements. WHENEVER provides tests for these conditions:

- An error occurred.
- A warning occurred.
- No rows were found.

When you specify a WHENEVER directive, the C compiler inserts statements that perform run-time checking after an SQL statement using the `sqlcode` variable.

[Table 9-1](#) lists the C compiler pseudocode generated to check `sqlcode` and the order in which the checks are made.

---

**Table 9-1. C Compiler Pseudocode for Checking the `sqlcode` Variable**

---

| Order | Condition  | Compiler Pseudocode                                                                    |
|-------|------------|----------------------------------------------------------------------------------------|
| 1     | NOT FOUND  | <code>if (sqlcode == 100) action-specification;</code>                                 |
| 2     | SQLERROR   | <code>if (sqlcode &lt; 0) action-specification;</code>                                 |
| 3     | SQLWARNING | <code>if (sqlcode &gt; 0) &amp;&amp; (sqlcode != 100)<br/>action-specification;</code> |

---

*action-specification* is one of:

```
CALL :host-identifier ;
GOTO :host-identifier ;
GO TO :host-identifier;
CONTINUE ;
```

When more than one WHENEVER condition applies to an SQL statement, NonStop 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 CALL :row_not_found;
EXEC SQL WHENEVER SQLERROR CALL :sql_error;
EXEC SQL WHENEVER SQLWARNING CALL :sql_warning;
...
```

---

**Note.** NonStop 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. To execute a different routine when an error occurs, specify a new WHENEVER directive with a different CALL routine.

For example, to insert a new row only when a row is not found, specify a new WHENEVER directive as follows:

```
EXEC SQL WHENEVER NOT FOUND CALL :insert_row;
```

The new WHENEVER directive remains in effect until it is disabled or changed.

- If a WHENEVER directive is coded in a function, the directive remains in effect outside of the function even if the scope of the function is no longer valid. Therefore, if you do not want the directive to remain in effect, disable it at the end of the function as described following.
- A program's order includes any files copied into the program using an include directive. If a copied file contains a WHENEVER directive, that directive remains in effect following the include directive.
- A WHENEVER directive does not affect SQL statements if they appear in the program before the WHENEVER directive.
- If you are debugging a program and you use a WHENEVER directive to call an error handling procedure, you might need to save the `sqlcode` value in a local variable within the error handling procedure. Each subsequent SQL statement resets `sqlcode`, and you might lose a value you need to debug the program.

## Enabling and Disabling the WHENEVER Directive

You can enable and disable the WHENEVER directive for different parts of a program. For example, you might want to handle SQL errors by checking the `sqlcode` variable after an SQL statement instead of using WHENEVER SQLERROR.

This example shows how to enable and disable the WHENEVER directive:

```
EXEC SQL
  WHENEVER SQLERROR CALL :err_func; /* enables checking */

...
EXEC SQL
  WHENEVER SQLERROR;                /* disables checking */
```

## 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 runs a statement that generates an error.
- The SQLWARNING condition runs a statement that generates a warning.
- The NOT FOUND condition runs a statement that generates a NOT FOUND condition.

To avoid these situations, disable the appropriate WHENEVER directive for the part of the program that handles the condition. [Example 9-2](#) on page 9-9 enables and disables the WHENEVER directive.

## Using the CALL Format

To use the CALL format to execute an error handling function, specify the WHENEVER directive globally and follow it with a forward declaration of the error handling functions. Also, ensure that each error handling function is accessible from all SQL statements affected by the WHENEVER directive.

## Using the GOTO Format

To use the GOTO (or GO TO) format, specify the WHENEVER directive at the beginning of the function containing the GOTO format and disable it at the end of the function. This example enables and disables the WHENEVER directive:

```
void func(void)
{
    EXEC SQL WHENEVER SQLERROR GOTO :error_handler;

    /* error_handler function */
    ...

    EXEC SQL WHENEVER SQLERROR; /* disable WHENEVER */
}
```

---

**Example 9-2. Enabling and Disabling the WHENEVER Directive**

```

EXEC SQL WHENEVER SQLERROR CALL :error_handler;

void fred(short i, short j, short k)
{
    EXEC SQL SELECT ...;
    EXEC SQL SELECT ...;
    EXEC SQL SELECT ...;
}
void ginger(short i, short j, short k)
{
    EXEC SQL SELECT ...;
    EXEC SQL SELECT ...;
    EXEC SQL SELECT ...;
}
/* reset SQLERROR checking while in error handler */
EXEC SQL WHENEVER SQLERROR;
void error_handler(void)
{
    EXEC SQL SELECT...;
    EXEC SQL SELECT...;
    EXEC SQL SELECT...;
}
/* enable SQLERROR checking */
EXEC SQL WHENEVER SQLERROR CALL :error_handler;

int main(void)
{
    fred();
    ginger();
    EXEC SQL INSERT...;
}

```

---

**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, specify a corresponding indicator variable and test the result of the indicator variable. Otherwise, NonStop SQL/MP returns an error specifying that no indicator variable was provided rather than the “not-found” condition. A WHENEVER NOT FOUND directive does not detect this condition.

## Example of Using WHENEVER Directives

The code in [Example 9-3](#) inserts two column values into the PARTS table and checks for errors and warnings using WHENEVER directives. Within the INSERT statement, the WHENEVER SQLERROR directive is processed first. This directive has a higher precedence, although the WHENEVER SQLWARNING directive is specified first in the source code.

---

### Example 9-3. Using the WHENEVER Directive (page 1 of 2)

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 315;

#define MAX_PARTDESC 19
EXEC SQL BEGIN DECLARE SECTION;
    struct
    { short in_partnum;
      long in_price;
      char in_partdesc[MAX_PARTDESC];
    } in_parts_rec;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;    /* For use with SQLCADISPLAY */

short sqlcode;

short ix;    /* Loop counter for blank padding in_partdesc */

/* Specify WHENEVERs globally for errors and warnings: */

EXEC SQL WHENEVER SQLWARNING CALL :handle_warnings;
EXEC SQL WHENEVER SQLERROR CALL :handle_errors;

/* Forward declare error handling code: */

void handle_warnings(void);
void handle_errors(void);
```

---

---

**Example 9-3. Using the WHENEVER Directive** (page 2 of 2)

---

```

int main(void)
{
    /* Begin TMF transaction: */
    EXEC SQL BEGIN WORK;
        in_parts_rec.in_partnum = 4120;
        in_parts_rec.in_price   = 6000000;
        strcpy (in_parts_rec.in_partdesc, "V8 DISK OPTION");

    /* Blank pad in_partdesc.  "V8 DISK OPTION" occupies */
    /* positions 0 through 13; start blank padding at    */
    /* position 14:                                     */

        for (ix = 14; ix <
; ix++)
            in_parts_rec.in_partdesc[ix] = ' ';
    ...

    /* Do an SQL INSERT into the parts table: */

        EXEC SQL
            INSERT INTO =parts (partnum, price, partdesc)
            VALUES ( :in_parts_rec.in_partnum,
                    SETSCALE (:in_parts_rec.in_price, 2),
                    :in_parts_rec.in_partdesc);

    /* End TMF transaction:                               */
        EXEC SQL COMMIT WORK;
    }

void handle_errors(void)
{
    SQLCADISPLAY( (short *) &sqlca);
    exit(EXIT_FAILURE);
}

void handle_warnings(void)
{
    warning_sum++;
    SQLCADISPLAY( (short *)&sqlca, , , 'N', 'Y');
}

```

---

## Returning Information From the SQLCA Structure

NonStop SQL/MP returns run-time information, including errors and warnings, for the most recently run SQL statement to the SQL communications 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

To declare an SQLCA structure, specify the INCLUDE SQLCA directive using this syntax (if you do not first specify the INCLUDE STRUCTURES directive, NonStop SQL/MP generates version 2 structures by default):

```
EXEC SQL INCLUDE SQLCA;
```

**Table 9-2. C Identifiers Generated by the INCLUDE SQLCA Directive**

| Name              | Value | Description                            |
|-------------------|-------|----------------------------------------|
| SQLCA_EYE_CATCHER | CA    | Eye-catcher value                      |
| SQLCA_LEN         | 430   | Length in bytes of the SQLCA structure |

For a description of the information returned to the SQLCA structure, see the SQLCAGETINFOLIST procedure item codes in [Section 5, SQL/MP System Procedures](#).

### Using System Procedures With the SQLCA Structure

[Table 9-3](#) describes the SQL system procedures you can use to retrieve and display information from the SQLCA structure. To call these procedures, a program must include declarations from the `cextdec` header file.

**Table 9-3. System Procedures for the SQLCA Structure**

| System Procedure | Description                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------------------|
| SQLCADISPLAY     | 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 |
| SQLCATOBUFFER    | 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 about these SQL/MP system procedures, see [Section 5, SQL/MP System Procedures](#).



# Returning Performance and Statistics Information

NonStop SQL/MP returns performance and statistics information to the SQL statistics area (SQLSA) after the execution of these DML statements:

- An INSERT, UPDATE, or 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, NonStop SQL/MP returns information in the SQLSA structure for these statements:

- Each PREPARE statement, including 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.

An SQL statement resets the SQLSA values. If you use an SQLSA value elsewhere in a program, save the value in a variable immediately after the statement runs. 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 runs. (In some cases, you can also declare more than one SQLSA structure.)

## Declaring the SQLSA Structure

To declare an SQLSA structure, specify the INCLUDE SQLSA directive using this syntax:

```
EXEC SQL INCLUDE SQLSA;
```

## Using the SQLSA Structure

Follow these guidelines when you use the information in an SQLSA structure:

- To generate a version 300 or later SQLSA structure, first specify an INCLUDE STRUCTURES directive with the version of the SQLSA structure you require. Otherwise, NonStop 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 runs (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.

[Table 9-4](#) describes the C identifiers generated by the INCLUDE SQLSA directive. Always use the symbolic names for these identifiers rather than the actual values, because the values can change in a new RVU.

---

**Table 9-4. C Identifiers Generated by the INCLUDE SQLSA Directive**

| Name              | Value             | Description                                                                                                                                                                                                    |
|-------------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLSA_EYE_CATCHER | SA                | Eye-catcher value. Use <code>SQLSA_EYE_CATCHER</code> to initialize the <code>eye_catcher</code> field in the SQLSA structure.                                                                                 |
| SQLSA_LEN         | 838<br>or<br>1790 | Length in bytes of the SQLSA structure. A version 330 or later SQLSA structure is 1790 bytes; older SQLSA structures are 838 bytes. Use <code>SQLSA_LEN</code> to allocate extra copies of an SQLSA structure. |

---

The native mode C compiler also generates this compiler pragma for version 330 or later SQLSA structures:

```
#pragma fieldalign cshared2 SQLSA_TYPE_R330 DML_TYPE_R330
STATS_TYPE_R330 PREPARE_TYPE_R330
```

[Example 9-4](#) on page 9-15 shows the layout of a version 300 through 325 SQLSA structure (length is 838 bytes), whereas [Example 9-5](#) on page 9-16 shows the layout of a version 330 (or later) SQLSA structure (length is 1790 bytes).

(For the version 1 and version 2 SQLSA structures, see [Appendix D, Converting C Programs](#).)

---

**Example 9-4. Version 300-325 SQLSA Structure**

```

struct SQLSA_TYPE
{
    char    eye_catcher[2];
    short version;
    union
    {
        struct DML_TYPE
        {
            short num_tables;
            struct STATS_TYPE
            {
                char table_name[24];
                long records_accessed;
                long records_used;
                long disc_reads;
                long messages;
                long message_bytes;
                short waits;
                short escalations;
                char sqlsa_reserved[4];
            } stats[16];
        } dml;
        struct PREPARE_TYPE
        {
            short input_num;
            short input_names_len;
            short output_num;
            short output_names_len;
            short name_map_len;
            short sql_statement_type;
            long  output_collations_len;
        } prepare;
    } u;
} sqlsa;

```

---

---

**Example 9-5. Version 330 (or later) SQLSA Structure**

```

struct SQLSA_TYPE_R330
{
    char    eye_catcher[2];
    short version;
    union
    {
        struct DML_TYPE_R330
        {
            short num_tables;
            long long master_executor_elapsed_time;
            long long total_esp_cpu_time;
            long long total_sortprog_cpu_time;
            char filler[32];
            struct STATS_TYPE_R330
            {
                char table_name[24];
                long long records_accessed;
                long long records_used;
                long long disc_reads;
                long long messages;
                long long message_bytes;
                long waits;
                long escalations;
                short vsbb_write;
                short vsbb_flushed;
                char filler[32];
            } stats[16];
        } dml;
        struct PREPARE_TYPE_R330
        {
            short input_num;
            short input_names_len;
            short output_num;
            short output_names_len;
            short name_map_len;
            short sql_statement_type;
            long output_collations_len;
        } prepare;
    } u;
} sqlsa_r330;

```

---

**Table 9-5. SQLSA Structure Fields** (page 1 of 2)

| Field Name                                | Description                                                                                                                                                                                                     |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>eye_catcher</code>                  | Identification field. Set <code>eye_catcher</code> to <code>SQLSA_EYE_CATCHER</code> .                                                                                                                          |
| <code>version</code>                      | Current product version of SQLSA. (Subsequent NonStop SQL/MP PVUs can change this value.)                                                                                                                       |
| <code>dml</code>                          | Structure for the return of statistics after the execution of a DML statement.                                                                                                                                  |
| <code>num_tables</code>                   | Number of tables accessed by a DML statement; maximum is 16.                                                                                                                                                    |
| <code>master_executor_elapsed_time</code> | CPU time in microseconds used by the master Executor process. Applies only to a version 330 or later SQLSA structure.                                                                                           |
| <code>total_esp_cpu_time</code>           | Total CPU time in microseconds used by all Executor Server Processes (ESPs). Applies only to a version 330 or later SQLSA structure.                                                                            |
| <code>total_sortprog_cpu_time</code>      | Total CPU time in microseconds used by all SORTPROG processes. Applies only to a version 330 or later SQLSA structure.                                                                                          |
| <code>stats</code>                        | Array containing <code>num_tables</code> valid entries, one for each table accessed.                                                                                                                            |
| <code>table_name</code>                   | Guardian internal file name of the table accessed.                                                                                                                                                              |
| <code>records_accessed</code>             | Number of records accessed in the corresponding table.                                                                                                                                                          |
| <code>records_used</code>                 | Number of records altered or returned.                                                                                                                                                                          |
| <code>disc_reads</code>                   | Number of disk reads and writes.                                                                                                                                                                                |
| <code>messages</code>                     | Number of messages sent to the disk process.                                                                                                                                                                    |
| <code>message_bytes</code>                | Number of bytes sent in all the messages sent to the disk process.                                                                                                                                              |
| <code>waits</code>                        | Number of lock waits or time outs.                                                                                                                                                                              |
| <code>escalations</code>                  | Number of times record locks are escalated to file locks.                                                                                                                                                       |
| <code>sqlsa_reserved</code>               | Reserved.                                                                                                                                                                                                       |
| <code>vsbb_write</code>                   | True (-1) if a VSBB write was used. Otherwise, false (0).                                                                                                                                                       |
| <code>vsbb_flushed</code>                 | True (-1) if the VSBB buffer was flushed. Otherwise, false (0).                                                                                                                                                 |
| <code>prepare</code>                      | Structure for the return of statistics for a PREPARE statement. Applies only to dynamic SQL statements. A program can use this information to allocate the buffers required to describe the prepared statement. |

**Table 9-5. SQLSA Structure Fields** (page 2 of 2)

| Field Name             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|---|---------------|-----------------------|---|--------|-----------------------|---|--------|-----------------------|---|--------|--------------------|---|---------------|------------------------|---|------------------------|--------------------|---|------------------------------|--------------------|---|----------------|
| input_num              | Number of input parameters in the prepared statement.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| input_names_len        | Length of the buffer required to contain names of input parameters.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| output_num             | Number of output variables (host variables or SELECT columns) in the prepared statement.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| output_names_len       | Length of buffer required to contain names of output variables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| name_map_len           | Reserved.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| sql_statement_type     | Statement being prepared (name, value, and type): <table><tr><td>_SQL_STATEMENT_SELECT</td><td>1</td><td>Cursor SELECT</td></tr><tr><td>_SQL_STATEMENT_INSERT</td><td>2</td><td>INSERT</td></tr><tr><td>_SQL_STATEMENT_UPDATE</td><td>3</td><td>UPDATE</td></tr><tr><td>_SQL_STATEMENT_DELETE</td><td>4</td><td>DELETE</td></tr><tr><td>_SQL_STATEMENT_DDL</td><td>5</td><td>DDL statement</td></tr><tr><td>_SQL_STATEMENT_CONTROL</td><td>6</td><td>Run-time CONTROL TABLE</td></tr><tr><td>_SQL_STATEMENT_DCL</td><td>7</td><td>LOCK, UNLOCK, FREE RESOURCES</td></tr><tr><td>_SQL_STATEMENT_GET</td><td>8</td><td>GET VERSION...</td></tr></table> | _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... |
| _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...               |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
|                        | To use these declarations, copy the <code>sqlh</code> header file using an <code>#include</code> directive.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |
| output_collations_len  | Length of the output collation buffer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                              |   |               |                       |   |        |                       |   |        |                       |   |        |                    |   |               |                        |   |                        |                    |   |                              |                    |   |                |

# 10 Dynamic SQL Operations

Dynamic SQL allows a host-language program to construct, compile, and run all or part of an SQL statement at run time. A dynamic SQL program uses a character host variable as a placeholder for the SQL statement, which is usually unknown or incomplete until run time. To construct the dynamic SQL statement in the host variable, the program usually requires some input from a user at a terminal or workstation.

## Uses for Dynamic SQL

Dynamic SQL programs can be useful in these situations:

- **New user interface**—You need to develop an interactive design for a specific user. For example, you might want to provide a graphical user interface (GUI) or restrict the SQL commands a user can run.

A dynamic SQL program can be similar to SQLCI, requiring the user to know SQL syntax to formulate a complete SQL statement. A dynamic SQL program can also prompt the user for input, so that the user does not have to know any SQL syntax. If the statement requires input parameters, the program can also prompt the user for these values. The program can then construct the SQL statement by concatenating these values to SQL syntax elements. For example, a program might construct an entire SQL statement or only part of a statement, such as a WHERE clause.

- **Restricted access to data**—You want to restrict the access to specific columns in a table. For example, your program might include a dynamic SELECT statement that accesses specific columns in a table but not other columns (such as the columns for an employee's salary or home phone number).
- **Client-server support**—You need to develop a server that receives requests from client applications. For example, an application on a personal computer wants to access an SQL/MP database. The PC application formulates an SQL statement and sends it to your program over a communications protocol. Your program constructs, compiles, and runs the dynamic SQL statement, and then sends the results back to the PC application. (An example of a server that uses dynamic SQL is the HP NonStop ODBC Server.)

# Dynamic SQL Statements

You can perform most of the same operations using dynamic SQL statements that you can perform with static SQL statements, including DDL, DML, and DCL statements and SQL cursors. [Table 10-1](#) summarizes the dynamic SQL statements that you can use in a C program.

---

**Table 10-1. Dynamic SQL Statements**

| Statement         | Description                                                                                                                                                                                                                                                    |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DESCRIBE INPUT    | Returns information about input parameters associated with a prepared SQL statement.                                                                                                                                                                           |
| DESCRIBE          | Returns information about output variables (usually SELECT columns) associated with a prepared SQL statement.                                                                                                                                                  |
| PREPARE           | Dynamically compiles an SQL statement stored in a host variable and associates the prepared statement with a statement name (an SQL identifier) or a host-variable name.                                                                                       |
| EXECUTE           | Runs a prepared SQL statement.                                                                                                                                                                                                                                 |
| EXECUTE IMMEDIATE | Compiles and runs an SQL statement stored in a host variable.                                                                                                                                                                                                  |
| DECLARE CURSOR    | Defines an SQL cursor and associates the cursor with a SELECT statement.                                                                                                                                                                                       |
| OPEN              | Opens an SQL cursor: Runs the associated SELECT statement and positions the cursor before the first row specified by the SELECT statement so that subsequent FETCH statements can retrieve data. Optional USING clause provides values for dynamic parameters. |
| FETCH             | Positions an SQL cursor at the next row of the result table defined by the associated SELECT statement and then retrieves data into host variables.                                                                                                            |
| RELEASE           | Deallocates space in the host-language program for a dynamic SQL statement prepared from a host variable.                                                                                                                                                      |
| CLOSE             | Closes an SQL cursor and frees the result table defined by the associated SELECT statement.                                                                                                                                                                    |

---

These statements are described following. For the syntax of each statement, see the *SQL/MP Reference Manual*.



# Dynamic SQL Features

## SQLDA Structure, Names Buffer, and Collation Buffer

NonStop SQL/MP uses the SQLDA structure to return information about input parameters and output variables in dynamic SQL statements. The SQLDA structure also provides a pointer to these buffers:

- Names buffer—Receives the names of input parameters or output variables
- Collation buffer—Receives copies of any collations used by columns in the query

You can use the SQLDA structure in these statements:

- A DESCRIBE INPUT statement to get 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 fill a cursor with rows from an SQL table
- The USING DESCRIPTOR clause of an EXECUTE statement to run a dynamic SQL statement

## Declaring the SQLDA Structure

To declare an SQLDA structure, use the INCLUDE SQLDA directive as follows:

```
INCLUDE SQLDA ( sqlda-name [ , sqlvar-count ]
               [ , names-buffer, max-name-length ]
               [ , release-option ]
               [ , CPRULES collation-buffer, max-collation-size ] ) ;
```

*sqlda-name*

is the SQLDA structure name; it must follow the conventions for a C identifier.

*sqlvar-count*

is the number of input parameters (plus indicator parameters) for which you expect to specify values, or the number of columns for which you expect to receive output values. The C compiler creates a separate SQLVAR structure within the SQLDA structure for each parameter or column.

The default for *sqlvar-count* is 1.

*names-buffer*

is the SQLDA names buffer; it must follow the conventions for a C identifier.

*max-name-length*

is the maximum number of bytes you expect in a parameter name or column name to be returned in a DESCRIBE or DESCRIBE INPUT statement. A qualified column name can be from 1 to 30 bytes long and is in this format:

*table-name.column-name*

A parameter name is an SQL identifier with a maximum of 30 bytes.

*release-option*

specifies the version of the SQLDA structure generated by the C compiler. RELEASE1 specifies SQL/MP version 1, and RELEASE2 specifies SQL/MP version 2. If *release-option* specifies a version other than the default for the system, the C compiler appends \_R1 or \_R2 to the SQLDA names and identifiers.

---

**Note.** Although version 300 (and later) C compilers allow the RELEASE1 and RELEASE2 options, HP might not support these options in a future RVU. If you are using a version 300 (or later) C compiler to generate version 1 or version 2 data structures, replace the RELEASE1 or RELEASE2 option with the VERSION 1 or VERSION 2 option of the INCLUDE STRUCTURES directive.

---

CPRULES

is a required keyword if you specify a collation buffer.

*collation-buffer*

is a host variable specifying the name of the collation buffer. The COLLATIONS INTO clause of the DESCRIBE statement allows you to return collations to *collation-buffer*.

*max-collation-size*

is the maximum number of bytes you expect for any one collation.

[Table 10-2](#) describes the C identifiers generated by an INCLUDE SQLDA directive. Always use the symbolic names rather than the actual values because the values can change in a new RVU.

---

**Table 10-2. C Identifiers Generated by the INCLUDE SQLDA Directive**

| Name                    | Value | Description                                                                                                                                                                                                                                                |
|-------------------------|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLDA_EYE_CATCHER       | D1    | Eye-catcher value. Use this identifier to initialize the <code>eye_catcher</code> field in the SQLDA structure:<br><br><pre>strncpy(<i>sqlda</i>.eye_catcher,         SQLDA_EYE_CATCHER, 2) ;</pre> where <i>sqlda</i> is the name of the SQLDA structure. |
| SQLDA_HEADER_LEN        | 4     | The length in bytes of the SQLDA structure header fields <code>eye_catcher</code> and <code>num_entries</code> .                                                                                                                                           |
| SQLDA_SQLVAR_LEN        | 24    | The length in bytes of one SQLVAR entry.                                                                                                                                                                                                                   |
| SQLDA_NAMESBUF_OVHD_LEN | 11    | The overhead length in bytes added to the names buffer. This overhead is the length field (2 bytes), table name (8 bytes), and period separator (1 byte).                                                                                                  |

---

[Table 10-3](#) describes each field in a version 315 (or later) SQLDA structure.

---

**Table 10-3. SQLDA Structure Fields** (page 1 of 2)

| Field Name               | Description                                                                                                                                                                                       |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>eye_catcher</code> | An identifying field that a program must initialize. NonStop SQL/MP does not return a value to <code>eye_catcher</code> .                                                                         |
| <code>num_entries</code> | Number of input or output parameters the SQLDA structure can contain.                                                                                                                             |
| <code>sqlvar</code>      | Group item that describes input parameters or database columns. The DESCRIBE INPUT and DESCRIBE statements return one <code>sqlvar</code> entry for each input parameter or each output variable. |
| <code>data_type</code>   | Data type of the parameter. For the <code>data_type</code> values, see <a href="#">Table 10-4</a> on page 10-8.                                                                                   |

---

**Table 10-3. SQLDA Structure Fields** (page 2 of 2)

| Field Name             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>data_len</code>  | <p>The value depends on the data type:</p> <p>Fixed-length character      Number of bytes in the string.</p> <p>Variable-length character   Maximum number of bytes in the string.</p> <p>Decimal numeric              Bits 0:7 specify the decimal scale.</p> <p>Bits 8:15 specify the byte length of the item.</p> <p>Binary numeric                Bits 0:7 specify the decimal scale.</p> <p>Bits 8:15 specify the byte length of the item (2, 4, or 8).</p> <p>Date-time or INTERVAL      Bits 0:7 specify the range of the field. For these values, see <a href="#">Table 10-5</a> on page 10-10.</p> <p>Bits 8:15 specify the storage size of the item.</p> |
| <code>precision</code> | <p>The precision value depends on the data type:</p> <p>Binary numeric                Numeric precision.</p> <p>Date-time or INTERVAL      Bits 0:7 specify the leading field precision.</p> <p>Bits 8:15 specify the fraction precision.</p> <p>If the FRACTION field is not included, bits 8:15 are 0.</p> <p>Character and VARCHAR      Character set ID. For the <code>precision</code> values, see <a href="#">Table 10-5</a> on page 10-10.</p>                                                                                                                                                                                                              |
| <code>null_info</code> | <p>For input parameters, <code>null_info</code> is a negative integer if the column permits null values.</p> <p>For output columns, <code>null_info</code> is a negative integer if the row returned is null.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>var_ptr</code>   | <p>Extended address of the actual data (value of input parameter or database column). NonStop SQL/MP does not return a value to <code>var_ptr</code>. A program must initialize <code>var_ptr</code> to point to the input and output data buffers.</p>                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>ind_ptr</code>   | <p>Address of a flag that indicates whether a parameter or column is null. For input parameters, a program sets the <code>ind_ptr</code> location to -1 if the user entered a null value.</p> <p>For output columns, NonStop SQL/MP sets the <code>ind_ptr</code> location to -1 if the column value is null.</p> <p>A program that does not process null values should set <code>ind_ptr</code> to an invalid address.</p>                                                                                                                                                                                                                                        |
| <code>cprl_ptr</code>  | <p>For input columns, <code>cprl_ptr</code> is not set.</p> <p>For output columns, <code>cprl_ptr</code> contains the address of the collation used by the column if a collation was used.</p> <p>If a collation was not used for the output column, <code>cprl_ptr</code> contains a negative integer.</p>                                                                                                                                                                                                                                                                                                                                                        |

[Example 10-1](#) shows a version 315 SQLDA structure, names buffer, and collation buffer. For version 1 and 2 SQLDA structures, see [Appendix D, Converting C Programs](#).

---

### Example 10-1. SQLDA Structure and Buffers

```
struct SQLDA_TYPE
{
    char   eye_catcher[2];
    short  num_entries;
    struct SQLVAR_TYPE
    {
        short data_type;
        short data_len;
        short precision;
        short null_info;
        long   var_ptr;
        long   ind_ptr;
        long   cprl_ptr;
        long   reserved;
    } sqlvar[sqlvar-count];
} sqlda-name;

char names_buffer [name-string-size];
char collation_buffer [collation-buffer-length];
```

---

## Calculating the Lengths of the Names and Collation Buffers

NonStop SQL/MP returns a name to the names buffer as a VARCHAR item. The C compiler determines *length* in bytes of the names buffer using this formula:

$$length = (name-string-size + 11) * sqlvar-count$$

The 11 bytes added to *name-string-size* is derived from the length (2 bytes), table name (8 bytes), and period separator (1 byte).

Use the SQLDA\_NAMESBUF\_OVHD\_LEN identifier for this value.

NonStop SQL/MP returns a collation name to the collation buffer as a VARCHAR item. The C compiler determines the length in bytes of the collation buffer as follows:

$$collation-buffer-length = (max-collation-size + 4) * sqlvar-count$$

The 4 bytes added to *max-collation-size* is the length (len) field in the VARCHAR item. Use the SQLDA\_COLLBUF\_OVHD\_LEN identifier for this value.

## Using Declarations for the SQLDA Structure

HP provides declarations in the `sqlh` file that you can use for the SQLDA `data_type` and `precision` fields. Use the `#include` directive to copy these declarations into a C program. [Table 10-4](#) describes the declarations and values for the SQLDA `data_type` field.

**Table 10-4. SQLDA Data Type Declarations** (page 1 of 2)

| Value                                 | Declaration                    | Description                                      |
|---------------------------------------|--------------------------------|--------------------------------------------------|
| <b>Character Data Types (0 – 127)</b> |                                |                                                  |
| 0                                     | <code>_SQLDT_ASCII_F</code>    | Fixed-length single-byte character               |
| 1                                     | <code>_SQLDT_ASCII_F_UP</code> | Fixed-length single-byte character, upshifted    |
| 2                                     | <code>_SQLDT_DOUBLE_F</code>   | Fixed-length double-byte character               |
| 64                                    | <code>_SQLDT_ASCII_V</code>    | Variable-length single-byte character            |
| 65                                    | <code>_SQLDT_ASCII_V_UP</code> | Variable-length single-byte character, upshifted |
| 66                                    | <code>_SQLDT_DOUBLE_V</code>   | Variable-length double-byte character            |
| <b>Numeric Data Types (128 – 134)</b> |                                |                                                  |
| 130                                   | <code>_SQLDT_16BIT_S</code>    | 16-bit signed (signed SMALLINT)                  |
| 131                                   | <code>_SQLDT_16BIT_U</code>    | 16-bit unsigned (unsigned SMALLINT)              |
| 132                                   | <code>_SQLDT_32BIT_S</code>    | 32-bit signed (signed INT)                       |
| 133                                   | <code>_SQLDT_32BIT_U</code>    | 32-bit unsigned (unsigned INT)                   |
| 134                                   | <code>_SQLDT_64BIT_S</code>    | 64-bit signed (signed LARGEINT)                  |
| 140                                   | <code>_SQLDT_REAL</code>       | 32-bit floating point (REAL)                     |
| 141                                   | <code>_SQLDT_DOUBLE</code>     | 64-bit floating point (DOUBLE PRECISION)         |
| <b>Decimal Data Types (150 – 154)</b> |                                |                                                  |
| 150                                   | <code>_SQLDT_DEC_U</code>      | Unsigned DECIMAL                                 |
| 151                                   | <code>_SQLDT_DEC_LSS</code>    | DECIMAL, leading sign separate (not SQL type)    |
| 152                                   | <code>_SQLDT_DEC_LSE</code>    | ASCII DECIMAL, leading sign embedded             |
| 153                                   | <code>_SQLDT_DEC_TSS</code>    | DECIMAL, trailing sign separate (not SQL type)   |
| 154                                   | <code>_SQLDT_DEC_TSE</code>    | DECIMAL, trailing sign embedded (not SQL type)   |

**Table 10-4. SQLDA Data Type Declarations** (page 2 of 2)

| <b>Value</b>                                         | <b>Declaration</b> | <b>Description</b>   |
|------------------------------------------------------|--------------------|----------------------|
| <b>Date-Time and INTERVAL Data Types (192 – 212)</b> |                    |                      |
| 192                                                  | _SQLDT_DATETIME    | General Date-Time    |
| 195                                                  | _SQL_DTINT_Y_Y     | Year to Year         |
| 196                                                  | _SQL_DTINT_MO_MO   | Month to Month       |
| 197                                                  | _SQL_DTINT_Y_MO    | Year to Month        |
| 198                                                  | _SQL_DTINT_D_D     | Day to Day           |
| 199                                                  | _SQL_DTINT_H_H     | Hour to Hour         |
| 200                                                  | _SQL_DTINT_D_H     | Day to Hour          |
| 201                                                  | _SQL_DTINT_MI_MI   | Minute to Minute     |
| 202                                                  | _SQL_DTINT_H_MI    | Hour to Minute       |
| 203                                                  | _SQL_DTINT_D_MI    | Day to Minute        |
| 204                                                  | _SQL_DTINT_S_S     | Second to Second     |
| 205                                                  | _SQL_DTINT_MI_S    | Minute to Second     |
| 206                                                  | _SQL_DTINT_H_S     | Hour to Second       |
| 207                                                  | _SQL_DTINT_D_S     | Day to Second        |
| 208                                                  | _SQL_DTINT_F_F     | Fraction to Fraction |
| 209                                                  | _SQL_DTINT_S_F     | Second to Fraction   |
| 210                                                  | _SQL_DTINT_MI_F    | Minute to Fraction   |
| 211                                                  | _SQL_DTINT_H_F     | Hour to Fraction     |
| 212                                                  | _SQL_DTINT_D_F     | Day to Fraction      |

[Table 10-5](#) describes the declarations and values for the ranges of date-time and INTERVAL data types for the SQLDA `data_len` field.

**Table 10-5. SQLDA Date-Time and INTERVAL Declarations**

| Value | Declaration           | Description          |
|-------|-----------------------|----------------------|
| 1     | _SQL_DTINT_QUAL_Y_Y   | Year to Year         |
| 2     | _SQL_DTINT_QUAL_MO_MO | Month to Month       |
| 3     | _SQL_DTINT_QUAL_D_D   | Day to Day           |
| 4     | _SQL_DTINT_QUAL_H_H   | Hour to Hour         |
| 5     | _SQL_DTINT_QUAL_MI_MI | Minute to Minute     |
| 6     | _SQL_DTINT_QUAL_S_S   | Second to Second     |
| 7     | _SQL_DTINT_QUAL_F_F   | Fraction to Fraction |
| 8     | _SQL_DTINT_QUAL_Y_MO  | Year to Month        |
| 9     | _SQL_DTINT_QUAL_Y_D   | Year to Day          |
| 10    | _SQL_DTINT_QUAL_Y_H   | Year to Hour         |
| 11    | _SQL_DTINT_QUAL_Y_MI  | Year to Minute       |
| 12    | _SQL_DTINT_QUAL_Y_S   | Year to Second       |
| 13    | _SQL_DTINT_QUAL_Y_F   | Year to Fraction     |
| 14    | _SQL_DTINT_QUAL_MO_D  | Month to Day         |
| 15    | _SQL_DTINT_QUAL_MO_H  | Month to Hour        |
| 16    | _SQL_DTINT_QUAL_MO_MI | Month to Minute      |
| 17    | _SQL_DTINT_QUAL_MO_S  | Month to Second      |
| 18    | _SQL_DTINT_QUAL_MO_F  | Month to Fraction    |
| 19    | _SQL_DTINT_QUAL_D_H   | Day to Hour          |
| 20    | _SQL_DTINT_QUAL_D_MI  | Day to Minute        |
| 21    | _SQL_DTINT_QUAL_D_S   | Day to Second        |
| 22    | _SQL_DTINT_QUAL_D_F   | Day to Fraction      |
| 23    | _SQL_DTINT_QUAL_H_MI  | Hour to Minute       |
| 24    | _SQL_DTINT_QUAL_H_S   | Hour to Second       |
| 25    | _SQL_DTINT_QUAL_H_F   | Hour to Fraction     |
| 26    | _SQL_DTINT_QUAL_S_S   | Second to Second     |
| 27    | _SQL_DTINT_QUAL_S_F   | Second to Fraction   |
| 28    | _SQL_DTINT_QUAL_F_F   | Fraction to Fraction |



## Determining Character Set IDs From the `precision` Field

[Table 10-6](#) describes the character-set values that NonStop SQL/MP returns to the SQLDA `precision` field for CHAR and VARCHAR data types and the character-set declarations in the `sqlh` file that you can use in a C program:

**Table 10-6. SQLDA Character-Set IDs**

| Value | Declaration                         | Description                          |
|-------|-------------------------------------|--------------------------------------|
| 0     | <code>_SQL_CHARSETID_UNKNOWN</code> | A single-byte unknown character set. |
| 1     | <code>_SQL_CHARSETID_KANJI</code>   | Japanese (same as the Shift-JIS)     |
| 12    | <code>_SQL_CHARSETID_KSC5601</code> | Korean                               |
| 101   | <code>_SQL_CHARSETID_88591</code>   | ISO 8859/1                           |
| 102   | <code>_SQL_CHARSETID_88592</code>   | ISO 8859/2                           |
| 103   | <code>_SQL_CHARSETID_88593</code>   | ISO 8859/3                           |
| 104   | <code>_SQL_CHARSETID_88594</code>   | ISO 8859/4                           |
| 105   | <code>_SQL_CHARSETID_88595</code>   | ISO 8859/5                           |
| 106   | <code>_SQL_CHARSETID_88596</code>   | ISO 8859/6                           |
| 107   | <code>_SQL_CHARSETID_88597</code>   | ISO 8859/7                           |
| 108   | <code>_SQL_CHARSETID_88598</code>   | ISO 8859/8                           |
| 109   | <code>_SQL_CHARSETID_88599</code>   | ISO 8859/9                           |

For CHAR and VARCHAR parameters and variables, the `precision` field contains the character set ID. When a dynamic SQL statement runs, NonStop SQL/MP checks the `precision` field to ensure that the character-set ID matches the expected character set of the parameter or column, which is determined by the value in `COLUMNS.CHARACTERSET`.

If the character sets do not match, NonStop SQL/MP returns an error. However, if the program expects an unknown character set and the `CHARACTERSET` value for the parameter or column indicates a single-byte character set, NonStop SQL/MP does not return an error.

## Input Parameters and Output Variables

A parameter is a name in a dynamic SQL statement that serves as a place holder for a value substituted when the statement runs. Using a parameter, an SQL statement can be compiled without the input values. The input values are then substituted when the statement runs. The syntax for a parameter is shown in the *SQL/MP Reference Manual*.

Input parameters are specified in the statement as either a question mark (?) or a question mark plus a name (`?val`). An input parameter can appear in an SQL expression wherever a constant can appear. The program uses the `DESCRIBE INPUT`

statement with an input SQLDA structure to get information about the input parameters and obtain pointers to the input values.

NonStop SQL/MP returns data to a program through output variables. Output variables are user-specified areas in the program. Output variables can be host variables or individual data buffers to which the program (through the SQLDA structure) contains pointers. Output variables usually contain columns returned from a SELECT operation. A program uses the DESCRIBE statement to get information about the output variables.

This sequence shows a typical context for input parameters and output variables in dynamic SQL. If you know in advance which columns are likely to be selected, you can use this sequence:

```
strcpy ( hostvar, "SELECT empnum, salary FROM =employee \
WHERE salary > ?sal"); /* input parameter sal */

/* Blank pad the statement buffer, dynamically compile */
/* the statement, describe its variables, prompt the */
/* user and read in the value for sal, declare and */
/* open a cursor for the statement. */
...

EXEC SQL
  FETCH cursor INTO :enum, :sal; /* output variables */
                                /* :enum and :sal */
```

If you do not know in advance which columns to select, you can send the output values to data buffers the program allocated earlier and to which the program set up pointers. The pointers are in the SQLDA structure. In this case, the FETCH statement would look like this:

```
EXEC SQL
  FETCH cursor
    USING DESCRIPTOR :sqlda; /* SQLDA contains pointers */
                           /* to output data buffers. */
```

Internally, SQL execution is the same for both scenarios.

## 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 this UPDATE statement is stored in the host variable named `update_statement`:

```
UPDATE table SET col1 = ?a, col2= ?a, col3 = ?b
```

A PREPARE statement prepares the statement in `exec_statement` from the host variable `update_statement`:

```
PREPARE exec_statement FROM :update_statement; ...
```

To supply values for the UPDATE statement at run time, the program uses the two host variables `host_var1` and `host_var2`:

```
EXECUTE exec_statement USING :host_var1, :host_var2;
```

The value stored in `host_var1` is used for both instances of the parameter named ?a. The value stored in `host_var2` is used for the parameter named ?b. If you use three host variables, NonStop 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, and the value in the third host variable remains unused.

For example, in this statement, NonStop 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.

```
EXECUTE exec_statement
  USING :host_var1, :host_var2, :host_var3;
```

---

△ **Caution.** If you use the same parameter name more than once in a statement, NonStop SQL/MP gives each duplicate occurrence of the parameter the same data type, length, and other attributes as the first occurrence. As a result, 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 Loops

Parameters are often used when a dynamic SQL statement is run repeatedly with different input values. In this example, 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 TACL DEFINE named `=parts` represents the PARTS table.

1. A user enters this SQL statement:

```
UPDATE =parts SET price = ?p
```

2. The program copies the statement into the host variable named `intext`.
3. The program uses the PREPARE and DESCRIBE INPUT statements to return a description of the parameter in the input SQLDA structure (`in_sqlda`) and to get the name of the parameter in the input names buffer (`i_namesbuf`). The prepared statement is named `exec_stmt`.

```
EXEC SQL PREPARE exec_stmt FROM :intext;
EXEC SQL DESCRIBE INPUT exec_stmt INTO :in_sqlda
      NAMES INTO :i_namesbuf;
```

4. The program enters a loop to prompt 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 a buffer pointed to by in_sqlda */
/* Run the statement using each successive value      */

EXEC SQL EXECUTE exec_stmt USING DESCRIPTOR :in_sqlda;

/* end of loop */

```

## Using the Names Buffer for Parameter Values

When you use the names buffer to prompt the user for parameter values, the names buffer contains the names of parameters. When you use the names buffer to display column values, the names buffer contains the names of columns.

If you specify NAMES INTO in the DESCRIBE INPUT statement, the names buffer contains the names of parameters that you can use to prompt the user for parameter values. The data returned to the names buffer is in this form:

```
length_1  name_1      length_2  name_2  ...  length_n  name_n
```

where *name\_1* is the first name, *name\_2* the second name, and *name\_n* the last name.

The name length information is a 2-byte integer (SQL data type PIC S9(4) COMP, C data type int). 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. Expressions appear as a null string with a length of 0.

For the program to determine the names in the names buffer, 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 runs, the information for each parameter or column is in the SQLVAR array; the *var\_ptr* field contains the address of the length field for each parameter or column name 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 | Individual Entry Part        | Description                                                                                        |
|----------------|------------------------------|----------------------------------------------------------------------------------------------------|
| 04 ABCD        | 00000000000000100 <br> ABCD  | 2-byte length 4-character string with value = 4<br>4-character string                              |
| 06 ABCDE       | 00000000000000110 <br> ABCDE | 2-byte length 4-character string with value = 6<br>5-character string padded with 1 trailing blank |
| 00             | 00000000000000000 <br>       | 2-byte length with value = 0<br>Null string                                                        |

---

**Note.** If your program accepts null values for input parameters, the indicator parameter names are included in the names buffer. For more information, see [Handling Null Values in Input Parameters](#) on page 10-17.

---

To prompt the user with the parameter names in the input names buffer, you must read the length of the name and then position a pointer past the length field and onto the name.

[Example 10-2](#) uses the names buffer, prompts for input, and then reads parameter values entered by the user.

---

#### Example 10-2. Getting Parameter Values (page 1 of 2)

---

```
int request_invars( sqldaptr input_sqlda_ptr,
                   char *input_namesbuf_ptr )
{
    #define data_array_size 21
    /* size for numeric parameter value; maximum is 19      */
    /* digits plus sign byte plus null byte terminator      */
    /*
    int *len_ptr;          /* ptr to access length portion */
                        /* of names buffer information */
    int name_len;         /* number of bytes in a      */
                        /* parameter name            */
    int num_entries;      /* number of input parameters */
    int i;                /* loop counter              */
    char name_array[31];  /* for null-terminated name of */
                        /* the input parameter        */
    char data_array[data_array_size]; /* for a numeric value */
    int data_len;         /* number of bytes needed in  */
                        /* input value                 */
    int data_read;        /* number of bytes of input   */
                        /* value actually read         */
    char *lastchar;       /* last character read         */
                        /* for advancing pointer      */
    ...
    num_entries = input_sqlda_ptr->num_entries;
    printf("\nPlease provide values for the input
    parameters:/n);
    ...
}
```

---

---

**Example 10-2. Getting Parameter Values** (page 2 of 2)

---

```

for (i=0; i < num_entries; i++)
{
    /* Set pointer to the length prefix in names buffer:      */
    len_ptr = (int *)input_namesbuf_ptr;
    name_len = *len_ptr;

    /* Move pointer past the length prefix and onto a name: */
    input_namesbuf_ptr += 2; /* Store null-terminated parameter
name in name_array:      */
    if (name_len == 0 )
        name_array[0] = '\0'; /* Parameter had no name      */
    else
    {
        lastchar = input_namesbuf_ptr + (name_len - 1);
        if (*lastchar == ' ') /* last character is blank      */
            /* SQL inserts blanks to make      */
            /* the length fall on an even      */
            /* byte boundary if the name      */
            /* had an odd number of          */
            /* characters.                    */
        {
            strncpy (name_array, input_namesbuf_ptr, name_len - 1);
            name_array[name_len] = '\0';
        }
        else
        {
            strncpy (name_array, input_namesbuf_ptr, name_len);
            name_array[name_len] = '\0';
        }
    } /* end else; read and store named parameter      */

    /* Use a switch statement to check the data type, prompt*/
    /* the user for input, (using the parameter name in      */
    /* name_array), call a function to read the value          */
    /* input by the user (see sample program for code) .      */

    output_namesbuf_ptr = lastchar + 1;
} /* end of for loop */
}

```

---

## Null Values

The input and output SQLDA structures have the `null_info` and `ind_ptr` fields, which are used for handling null values. Your program accesses these fields in the SQLVAR array in the same way in which you access `var_ptr`.

`null_info` indicates whether the input parameter or output variable can contain a null value.

`ind_ptr` points to a flag that indicates whether the input parameter or output variable is null. If the parameter or output variable is not null, the `var_ptr` field specifies the value.

If you want all your parameters and output variables to handle null values, your program should access `ind_ptr` every time it accesses `var_ptr`.

## Handling Null Values in Input 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; for 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 non-null 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.

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 occurs when a null value is encountered.

After `DESCRIBE INPUT` runs and for each input parameter described in an `sqlvar` array in the input `SQLDA` structure, NonStop SQL/MP sets `null_info` to -1 if the input parameter in the prepared statement allows a null value (that is, if the prepared statement included a null indicator).

Your program then checks `null_info`. If `null_info` contains a -1 and you are allocating memory dynamically, you can now allocate two bytes of memory for a null indicator value, and then set `ind_ptr` to point to the memory. Allocate this memory at the same time you allocate memory for a possible nonnull parameter value.

If the user specifies a null value for the parameter, assign a -1 to the location pointed to by `ind_ptr`. NonStop SQL/MP checks this value and then transmits a null value for the parameter.

However, if the user does not enter a null value for the input parameter, you can assign a 0 to the location indicated by `ind_ptr`. NonStop SQL/MP checks `ind_ptr`, sees that `ind_ptr` indicates a nonnull value, and gets the parameter value from the location indicated by `var_ptr`.

## Handling Null Values in Output Variables

`DESCRIBE` sets `null_info` to -1 if the output variable can be null (that is, if the prepared statement includes a null indicator). If the value returned is null, NonStop 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 zero (0) or if `ind_ptr` is an invalid address.)

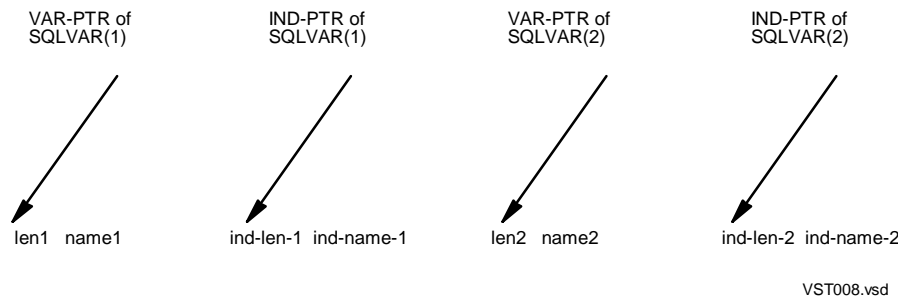
Your program must check `null_info` to determine whether the value returned can be null. If `null_info` contains a -1, then your program checks the location pointed to by `ind_ptr`. If that location contains a -1, then a null value was returned. If the location contains 0, then a nonnull value was returned and your program should get the value from the location pointed to by `var_ptr`.

## Null Values and the Names Buffer

If your program processes indicator parameters, the names of the indicator parameters are included in the names buffer after DESCRIBE INPUT runs. The `ind_ptr` field points to the length field for the first indicator parameter name in the names buffer. This behavior is parallel to that of `var_ptr` after DESCRIBE INPUT or DESCRIBE.

[Figure 10-1](#) is a diagram of the names buffer immediately after the DESCRIBE INPUT statement runs when indicator parameters are present for two parameters, where `len` is a two-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. Each instance of `ind_ptr` points to the length field for the corresponding indicator parameter name.

**Figure 10-1. DESCRIBE INPUT's Effect on Names Buffer**



Like input parameter and output variable names, indicator variable names are blank padded to even lengths.

When you are reading through the names buffer to prompt the user for parameter names, you might need to be aware of the indicator fields and perform tasks like the following:

1. Check the `null_info` field.
2. If `null_info` is -1, read the length field for the indicator.
3. Add this length field to the pointer or index to skip to the next name in the names buffer.

## Dynamic Allocation of Memory

A C program can dynamically allocate memory for input parameters and output variables at run time by following these steps:

1. Make sure that your program uses the large memory module (which is the default). The `XMEM` pragma causes the C compiler to use the large-memory model.
2. Declare a template for the `SQLDA` structure and names buffer using the `INCLUDE SQLDA` directive. Use a value of 1 for the `SQLDA` and names buffer sizes and



provide names for the SQLDA and names buffer as shown:

```
EXEC SQL INCLUDE SQLDA (dummy_da, 1, dummy_namesbuf, 1);
```

The INCLUDE directive generates the structure templates `sqlda_type` and `sqlvar_type`, which you can later use to allocate the memory. You might set up the pointers that will eventually point to that memory. For example:

```
typedef struct SQLDA_TYPE *sqldaptr;
sqldaptr input_sqlda_ptr, output_sqlda_ptr;
```

When the memory is allocated, `input_sqlda_ptr` points to the memory for the input SQLDA, and `output_sqlda_ptr` points to the memory for the output SQLDA. To access the SQLDA and names buffer, declare pointers and then pass the pointers to a function that allocates the memory as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct SQLDA_TYPE *sqldaptr;
    sqldaptr input_sqlda_ptr, output_sqlda_ptr;
    typedef char (*arrayptr) [1000];
    arrayptr input_namesbuf_ptr, output_namesbuf_ptr;
EXEC SQL END DECLARE SECTION;
```

To give the DESCRIBE INPUT and DESCRIBE statements a size to use before the memory is actually allocated, you must declare the names buffer to be an arbitrarily large size (this example uses 1000). Estimate a number that is greater than any possible size your names buffer could be. Otherwise, DESCRIBE INPUT and DESCRIBE might stop describing parameters or variables too soon.

3. Declare an SQLSA structure using the INCLUDE SQLSA directive:

```
EXEC SQL INCLUDE SQLSA;
```

4. Dynamically compile the SQL statement (`stmt1`) entered by the user using the PREPARE statement as shown in this example:

```
#define MAX_QUERY_SIZE 512
EXEC SQL BEGIN DECLARE SECTION;
    char statement_buffer[MAX_QUERY_SIZE + 1];
    ...
EXEC SQL END DECLARE SECTION;
    ...
printf("\nEnter a new SQL statement:\n");
/* Pass statement_buffer to a function that reads */
/* and parses the input (code is in sample program) */
    ...
EXEC SQL PREPARE stmt1 FROM :statement_buffer;
```

5. Use the information in the SQLSA structure to determine the number of input parameters and output variables in the statement.
6. Allocate space for the required number of SQLDA data structures to describe the parameters and output variables using the `malloc` function.
7. Allocate space for the values input to the program or output from the database, again using the `malloc` function.

## 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. These paragraphs provide some points to consider when you use cursors.

The order for executing statements to use a cursor with dynamic SQL is shown in this table:

| Operation                                                        | Description                                                                               |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| PREPARE <i>statement-name</i><br>FROM <i>:host-variable</i>      | Dynamically compiles the SELECT statement defining the cursor                             |
| Run DESCRIBE INPUT and DESCRIBE statements.                      | Retrieve information about the input and output parameters of the prepared SQL statement. |
| DECLARE <i>cursor-name</i> CURSOR<br>FOR <i>statement-name</i>   | Declares the dynamic cursor                                                               |
| OPEN <i>cursor-name</i><br>USING DESCRIPTOR <i>input-sqlda</i>   | Opens the cursor and gets parameter values from the input data buffer in the program      |
| FETCH <i>cursor-name</i><br>USING DESCRIPTOR <i>output-sqlda</i> | Retrieves data and outputs column values to output data buffer in the program             |
| Loop until "not-found" condition occurs.                         | Fetch data for all selected rows                                                          |
| CLOSE <i>cursor-name</i>                                         | Closes the cursor                                                                         |

Follow these guidelines when you declare and use a cursor:

- If you are using the COBOL85 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 4-24.
- 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 parameters entered at run time, use the USING DESCRIPTOR clause with the OPEN statement to provide the parameter values to SQL from an input location in the program's variable declarations. The input SQLDA describes the input location for each parameter. The DESCRIBE INPUT statement fills in the SQLDA SQLVAR entries, and your program sets the `var_ptr` fields and prompts the user for values for the parameters.

Use the USING DESCRIPTOR clause with the FETCH statement to write column values to an output location 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 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 uses 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.

```
strncpy (hostvar,
        "SELECT salary FROM =employee FOR UPDATE OF salary", 49);

EXEC SQL
    PREPARE stmt1 FROM :hostvar;

EXEC SQL
    DECLARE c1 CURSOR FOR stmt1;

EXEC SQL
    OPEN c1;

EXEC SQL
    FETCH c1 INTO :salvar;

EXEC SQL
    UPDATE =employee SET salary = salary * 1.20
    WHERE CURRENT OF c1;
```

## Using Statement and Cursor Host Variables

The DESCRIBE statement returns descriptions of output variables from previously prepared dynamic SQL statements. You can use statement and cursor host variables with the DECLARE CURSOR, PREPARE, OPEN, FETCH, and CLOSE statements. For each new statement and cursor name, store the name in the host variable before executing the statement. Thus, you code the statements only once.

[Example 10-3](#) shows the use of statement and cursor host variables. The program in this example is a server that does repetitive processing using a restricted set of operations. For example, the program might handle a SELECT statement for which the user can enter any of three different WHERE clauses. When the server is started, you might run the PREPARE, DESCRIBE INPUT, DESCRIBE, and DECLARE CURSOR statements once for each possible version of the statement.

---

### Example 10-3. Using Statement and Cursor Host Variables

```
#define MAX_STRGS 3
#define STRING_LEN 81
...

EXEC SQL BEGIN DECLARE SECTION;
char s_hostvar[STRING_LEN]; /* statement host variable*/
char c_hostvar[STRING_LEN]; /* cursor host variable */
char t[MAX_STRGS][STRING_LEN]; /* statements table */
char c[MAX_STRGS][STRING_LEN]; /* cursors table */
...
/* Store the statements in table t and the cursors in table c */
...
for (i = 1; i <= MAX_STRGS; i++)
{
    EXEC SQL PREPARE :s_hostvar FROM :t[i];
    EXEC SQL DESCRIBE INPUT :s_hostvar INTO :input_sqlda;
    /* Call a function to handle the input parameters */
    ...
    EXEC SQL DESCRIBE :s_hostvar INTO :output_sqlda;
    /* Call a function to handle the output variables */
    ...

    EXEC SQL DECLARE CURSOR :c[i] CURSOR FOR :s_hostvar;
}

```

You now have three cursor variables, one for each of your three possible statements. You set up the cursors only once, but your program can now use them repeatedly, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char cur[string_len]; /* Cursor local variable */
EXEC SQL END DECLARE SECTION;

/* Loop while not EOF: */

/* Read $RECEIVE */
...
/* Examine a flag in request message to determine which */
/* cursor to use, and assign appropriate value to cur:*/
/* c[0], c[1], or c[2] */
...

EXEC SQL OPEN :cur;
EXEC SQL FETCH :cur INTO column host variables;
EXEC SQL CLOSE :cur;

/* Call a function to display results */

/* End of loop WHILE not EOF */

```

---

# Developing a Dynamic SQL Program

## Specify the SQL Pragma

Specify the SQL pragma to indicate to the SQL compiler that your program contains embedded SQL statements. For information about the SQL pragma, see [Section 6, Explicit Program Compilation](#).

## Copy any External Declarations

Copy any required external declarations, including SQL system procedures and C header files:

```
#include <cextdecs (SQLCADISPLAY)> ;
#include <stdioh>;
#include <sqlh>;          /* data-type literals      */
#include <stdlibh>;        /* malloc and free    */
#include <stringh>;        /* strcpy, strncpy     */
```

## Declare the `sqlcode` Variable and Host Variables

Declare the `sqlcode` variable and any required host variables:

```
short sqlcode;

EXEC SQL BEGIN DECLARE SECTION;

char statement_buffer[MAX_STATEMENT_LENGTH+1];

EXEC SQL INVOKE =employee AS employee_struct;
struct employee_struct employee_row;

input_sqllda_ptr... /* pointer to input SQLDA      */
output_sqllda_ptr ... /* pointer to output SQLDA     */
input_namesbuf_ptr... /* pointer to input names buffer */
output_namesbuf_ptr... /* pointer to output names buffer */

...

EXEC SQL END DECLARE SECTION;
```

## Specify Any **WHENEVER** Directives

If you use **WHENEVER** directives for error handling, code them anywhere in your program. However, you must declare the error handling functions before you declare the directives.

```
EXEC SQL WHENEVER SQLERROR CALL :handle_error;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
```

For more information about the **WHENEVER** directive, see [Section 9, Error and Status Reporting](#).

## Specify the INCLUDE STRUCTURES Directive

Specify the INCLUDE STRUCTURES directive to indicate the version of SQL structures you plan to use:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 315;
```

For more information about the INCLUDE STRUCTURES directive, see [Section 9, Error and Status Reporting](#).

## Declare the SQLDA Structure and Names Buffer

Declare the SQLDA structure to generate a template to use later in the program:

```
EXEC SQL INCLUDE SQLDA (dummy_sqlda, 1, dummy_namesbuf, 1);
```

For more information about the INCLUDE SQLDA directive, see [SQLDA Structure, Names Buffer, and Collation Buffer](#) on page 10-3.

## Declare an SQLSA Structure

Declare an SQLSA structure using the INCLUDE SQLSA directive:

```
EXEC SQL INCLUDE SQLSA;
```

For more information about the INCLUDE SQLSA directive, see [Section 9, Error and Status Reporting](#).

## Process the Input Parameters

If the `input_num` field in the SQLSA structure is greater than 0 (zero), process the input parameters. Otherwise, skip these steps and go to [Read and Compile the SQL Statement](#) on page 10-25.

1. Get the length of the names buffer (for parameter names) from the `input_names_len` field in the SQLSA structure.
2. Allocate memory for the input SQLDA (and names buffer, if needed). [Example 10-4](#) on page 10-30 uses the function named `allocate_sqlda` to perform this step.
3. Initialize the SQLDA header fields (`SQLDA_EYE_CATCHER` is defined by the C compiler):

```
*input_sqlda_ptr.eye_catcher = SQLDA_EYE_CATCHER;
*input_sqlda_ptr.num_entries = sqlsa.u.prepare.input_num;
```

4. Specify a DESCRIBE INPUT statement to access input parameters:

```
EXEC SQL DESCRIBE INPUT :statement_name
      INTO :*input_sqlda_ptr
      NAMES INTO :*input_namesbuf_ptr;
```

5. Loop through the `sqlvar` in the input SQLDA structure. Loop  $n$  times, where  $n$  is the number of parameters from the `input_num` field. On each iteration of the loop:

- a. Check the `data_type` field and, if necessary, adjust the data type so that the C program can handle and reset `data_len` accordingly.
- b. Allocate an amount of memory equal to the `data_len` field for the parameter.
- c. Set the `var_ptr` field to point to the memory.

If you are not allocating memory dynamically, declare a variable for each input parameter value, and put the address of the variable in `var_ptr`.

If you know the number and data type of your input parameter values, you set only `data_type`, `data_len`, and `var_ptr`.

Some programs might check `data_type` and `data_len` when the actual values are obtained.

- d. If you are handling null values, check the `null_info` field and continue according to its value:

0      Do not allocate any memory.

-1      Allocate 2 bytes of memory for the indicator value.

If necessary, set `ind_ptr` to point to the memory allocated in Step c. (If you are not allocating memory dynamically, define a variable for the indicator and put its address in `ind_ptr`.)

6. Loop through the names buffer to read the corresponding name for each parameter and prompt the user for each value. Read each value into the corresponding occurrence in the input data buffer, according to the data type of the value. If the parameter can be null (`null_info` is -1) and the value entered was null, set `ind_ptr` to -1.

## Read and Compile the SQL Statement

1. Read the SQL statement you want to run. Normally, a user enters this statement from a terminal or workstation. After you construct the statement, pad the remainder of the buffer, including the null terminator position, with blanks.
2. Assign a statement name to the host variable (if necessary).

```
char statement_name[11];
strncpy ( statement_name, "stmt1", 2 );
```

3. Compile the SQL statement using the PREPARE statement:

```
EXEC SQL PREPARE :statement_name FROM :statement_buffer;
```

## Process the Output Variables

If the `output_num` field in the `SQLSA` structure is greater than zero (0) after the PREPARE statement runs, perform these steps:

1. Get the length of the output names buffer from `sqlsa.u.prepare.output_names_len`.
2. Call the `allocate_sqlda` function to allocate memory for the output SQLDA and the output names buffer, if needed.
3. Initialize the SQLDA header fields (SQLDA\_EYE\_CATCHER is defined by the C compiler):

```
*output_sqlda_ptr.eye_catcher = SQLDA_EYE_CATCHER;
*output_sqlda_ptr.num_entries = sqlsa.output_num;
```

4. Run a DESCRIBE statement to access the output variables:

```
EXEC SQL DESCRIBE :STATEMENT_NAME
      INTO :*output_sqlda_ptr
      NAMES INTO :*output_namesbuf_ptr;
```

5. Loop through the `sqlvar` array in the output SQLDA. Loop *n* times, where *n* is the number of columns from `sqlsa.u.prepare.output_num`. On each iteration of the loop:
  - a. Check the `data_type` field. If necessary, adjust the data type so the C program can handle and reset `data_len` accordingly.
  - b. Allocate memory equal to `data_len` for the output column.
  - c. Set `var_ptr` to point to the memory.
  - d. If you are not allocating memory dynamically, you would have declared a variable for each possible column value and put the address of the variable in `var_ptr`.
  - e. If you know the number and data type of the output column values, you set only `data_type`, `data_len`, and `var_ptr`.
  - f. Some programs might check `data_type` and `data_len` when the actual values are obtained.
  - g. If you are handling null values, check the `null_info` field and continue according to its value:
    - 0     Do not allocate any memory.
    - 1    Allocate two bytes of memory for the indicator variable.
  - h. If necessary, set `ind_ptr` to point to the memory allocated in the previous step. (If you are not allocating memory dynamically, define a variable for the indicator and put its address in `ind_ptr`.)
6. To show column headings (similar to SQLCI), loop through the names buffer to read the corresponding name for each column and display the column names.



## Perform the Database Request and Display the Values

Assign a name to the cursor host variable:

```
char cursor_name[11];
strncpy (cursor_name, "c1", 2);
```

If the statement is a SELECT statement, follow these steps:

1. Declare a cursor for the statement:

```
EXEC SQL DECLARE :cursor_name CURSOR FOR :statement_name;
```

2. Begin a TMF transaction:

```
EXEC SQL BEGIN WORK;
```

3. Open the cursor:

```
EXEC SQL OPEN :cursor_name
      USING DESCRIPTOR :*input_sqlda_ptr;
```

4. Execute a loop to fetch the values and display them:

```
EXEC SQL FETCH :cursor_name
      USING DESCRIPTOR :*output_sqlda_ptr;
```

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 Step 6 of [Process the Output Variables](#) on page 10-25.)

Handle null values as follows:

- If `null_info` is -1, check the value indicated by `ind_ptr`.
- If `ind_ptr` is also -1, display a character representing a null value (for example, a blank or zero). Otherwise, display the value indicated by `ind_ptr`.

If you are ignoring null values, display the value indicated by `var_ptr`.

5. Close the cursor:

```
EXEC SQL CLOSE :cursor_name;
```

If the statement is not a SELECT statement, follow these steps:

1. Begin a TMF transaction:

```
EXEC SQL BEGIN WORK;
```

2. Run the statement:

```
/* If there were input parameters: */
EXEC SQL EXECUTE :statement_name
      USING DESCRIPTOR :*input_sqlda_ptr;

/* If there were no input parameters: */
EXEC SQL EXECUTE :statement_name;
```

## 3. End the TMF transaction:

```
EXEC SQL COMMIT WORK;
```

4. Call the `free` function to deallocate the memory for the SQLDA structures and names buffers and for the values.

After the input statement is dynamically compiled with the PREPARE statement, the SQLSA structure contains this information:

- The `input_num` field is the number of input parameters in the statement. Use this information to determine how many parameter values to request from the user.
- The `input_names_len` field is the length of the buffer required to contain the names of the input parameters.
- The `output_num` field is the number of output variables in the statement. Use this information to determine how many column values to report.
- The `output_names_len` field is the length of the buffer required to contain the names of the output variables.
- The `sql_statement_type` field is the type of statement that was prepared, which can have these values:

|                                     |   |
|-------------------------------------|---|
| <code>_SQL_STATEMENT_SELECT</code>  | 1 |
| <code>_SQL_STATEMENT_INSERT</code>  | 2 |
| <code>_SQL_STATEMENT_UPDATE</code>  | 3 |
| <code>_SQL_STATEMENT_DELETE</code>  | 4 |
| <code>_SQL_STATEMENT_DDL</code>     | 5 |
| <code>_SQL_STATEMENT_CONTROL</code> | 6 |
| <code>_SQL_STATEMENT_DCL</code>     | 7 |
| <code>_SQL_STATEMENT_GET</code>     | 8 |

A program can use these values to allocate memory for the number of input parameters and output variables, and for the input and output names buffer length.

## Allocate Memory for the SQLDA Structures and Names Buffers

To allocate memory for the SQLDA structures and names buffers for the input and output variables, use the `malloc` function. The `malloc(n)` function allocates a block of memory, *n* bytes long, and returns the address of that block. The function returns a pointer to `void`, which is compatible with any pointer type. However, to enhance readability, specify the intended type using a cast operator. To include the `malloc` function in the `stdlibh` library, specify:

```
#include <stdlibh>
```

The program calls `malloc` and puts the values returned by `malloc` into pointers to the SQLDA structure and names buffer variables defined earlier in the program.

### Initialize the `eye_catcher` and `ind_ptr` Fields

When you allocate the SQLDA, you must explicitly initialize the `eye_catcher` and `ind_ptr` fields. You must initialize `ind_ptr` even if your program is not using indicator variables to handle null values.

When you issue `INCLUDE SQLDA` to create the SQLDA template, the C compiler creates a `#define` for `SQLDA_EYE_CATCHER`, which you then use to initialize the `eye_catcher` field:

```
sqlda_name.eye_catcher = SQLDA_EYE_CATCHER;
```

Initialize the `ind_ptr` and `var_ptr` fields for each SQLVAR entry to `NULL`:

```
for (i = 0; i < num_entries; i++)
    sqlda_name -> sqlvar[i].ind_ptr = NULL;
    sqlda_name -> sqlvar[i].var_ptr = NULL;
```

### Prepare to Allocate the SQLDA Structure and Names Buffer

In preparation for allocating memory to store the SQLDA structure, get the number of input parameters or output variables from the SQLSA structure. Similarly, to allocate memory for the names buffer, get the length of the input or output names buffer from the SQLSA structure.

```
num_input_vars = sqlsa.u.prepare.input_num;

if (num_input_vars > 0)
    allocate_sqlda(num_input_vars);
...
in_nameslen = sqlsa.u.prepare.input_names_len;

if (in_nameslen > 0)
    allocate_namesbuf(in_nameslen);
```

[Example 10-4](#) shows the `allocate_sqlda` function, which is also called to allocate the output SQLDA structure. This function initializes the `eye_catcher` and `ind_ptr` fields.

---

#### Example 10-4. Allocating the SQLDA Structure

```

/* in main code:                                     */
/*      typedef struct SQLDA_TYPE *sqldaptr;          */
/*
/* sqlda_type and sqlvar_type are generated by INCLUDE SQLDA
*/

sqldaptr allocate_sqlda (num_entries)
int num_entries;      /* number of input or output variables */
{
    sqldaptr sqlda_ptr; /* pointer to be returned          */
    int mem_reqd;        /* number of bytes required for SQLDA */
    short i;             /* loop counter                      */

    sqlda_ptr = NULL;

    mem_reqd = sizeof( struct SQLDA_TYPE ) +
        ( (num_entries - 1) * sizeof( struct SQLVAR_TYPE ) );
    ...
    /* call malloc to allocate memory (error checking omitted */
    sqlda_ptr = (sqldaptr)malloc (mem_reqd);

    sqlda_ptr->num_entries = num_entries;

    ...
    /* Initialize eye_catcher and ind_ptr          */
    ...
    /* return the pointer to newly allocated memory:      */
    return(sqlda_ptr);
}

```

---

To allocate memory for the names buffer, call `malloc` and pass `in_nameslen`. You specify an arbitrarily large size for the space required because SQL must have advance information about the space where to store the names. The program still allocates only the memory that is actually needed for the names, and SQL ignores any unused memory.

In this call, `input_namesbuf_ptr` is a pointer to the memory allocated for an input names buffer:

```

typedef char (*arrayptr) [1000];
...
if (in_nameslen > 0)
    input_namesbuf_ptr = (arrayptr) malloc(in_nameslen);

```

## Allocate Memory for the Values

After the descriptions of input parameters and output variables are specified, the program must allocate space for the actual values. The user might enter these values for input parameters, or the system might return them for columns (output variables). These paragraphs describe how to handle input parameters.

The program uses the DESCRIBE INPUT statement to fill in the SQLDA and names buffer with the descriptions of input parameters in the SQL statement. If you specify NAMES INTO, the names of the parameters are also returned in the names buffer. For a statement whose name was stored in a statement host variable, the DESCRIBE INPUT statement is as follows:

```
EXEC SQL
  DESCRIBE INPUT :statement_name
    INTO :*input_sqlda_ptr
    NAMES INTO :*input_namesbuf_ptr;
```

The DESCRIBE INPUT statement places the descriptions for parameters into the input SQLDA and the names of parameters into the location pointed to by `input_namesbuf_ptr`.

Immediately after DESCRIBE INPUT runs, the `var_ptr` field in the SQLDA points to the first entry 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 the DESCRIBE INPUT or DESCRIBE statement. 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.

The program can now allocate memory for the parameter values to be entered.

## Handle Scale

If your program must handle numeric values with scale, read the scale information from the input SQLDA structure. The DESCRIBE INPUT statement places this information in bits 0 through 7 of the `data_len` field in the SQLVAR array.

If you can ignore scale, you can set the `data_len` field to 0, causing data truncation. Otherwise, save the scale information and write a function to handle scale. Your program must check the data type of the values input to the program. For declarations that represent the data types that you can use in your program, see [Table 10-4](#) on page 10-8. To include declarations for these literals, use the `#include` directive to copy the `sqlh` file.

[Example 10-5](#) allocates memory for input parameter values. You can use the same code later to allocate memory for output variables.

---

### Example 10-5. Allocating Memory for Parameters and Columns

```
int setupvarbuffers (sqlda_ptr)
{
    sqldaptr sqlda_ptr; /* pointer to input or output SQLDA */
    {
        int num_entries; /* # of input parameters/output columns*/
        int mem_reqd;    /* buffer size to be allocated */
        int i;           /* loop counter */
        /* Pass the SQLDA pointer to a function to handle any */
        /* unsupported data types and scale (code omitted here).*/
        /* If you do not need to handle the scale, you can set */
        /* scale to 0. */
        num_entries = sqlda_ptr->num_entries;
        for (i = 0; i < num_entries; i++)
        {
            switch ( sqlda_ptr->sqlvar[i].data_type )
            {
                case _SQLDT_ASCII_F : /* char type */
                    mem_reqd = sqlda_ptr->sqlvar[i].data_len;
                    break;
                case _SQLDT_ASCII_V: /* varchar type */
                    mem_reqd = sqlda_ptr->sqlvar[i].data_len + 2;
                    break;
                ...
                /* For the numeric data types, either save the */
                /* scale information found in bits 0:7, or set */
                /* these bits to 0. Then extract the length */
                /* from bits 8:15: */
                case _SQLDT_16BIT_S:
                case _SQLDT_16BIT_U:
                ...
                /* Set scale information to zero: */
                sqlda_ptr->sqlvar[i].data_len =
                    sqlda_ptr->sqlvar[i].data_len & 0377;

                /* extract length from bits 8:15: */
                mem_reqd = sqlda_ptr->sqlvar[i].data_len & 0377;
                break;
                ... /* If data type is unsupported, return -1
            } /* end of switch statement; check data type */

            /* Allocate memory for the parameter or column; assign */
            /* address to the var_ptr field in the input SQLDA. */

            sqlda_ptr->sqlvar[i].var_ptr=(long) malloc (mem_reqd);

        } /* end of loop for memory for each parameter value */
        /* or column value */
    }
    return (0);
}
```

---

The program can now prompt the user for the input parameter values, set the pointer to the first SQLVAR element in the input SQLDA, and read through the SQLVAR array, storing each value the user enters into the appropriate position in memory.

## Allocate and Fill In Output Variables

To allocate space for output variables, you essentially perform the same set of operations described for input parameters except that the pointers point to the output SQLDA and names buffer. To get the descriptions of output variables into the output SQLDA, use the DESCRIBE statement instead of DESCRIBE INPUT:

```
EXEC SQL DESCRIBE :statement_name
      INTO :*output_sqlda_ptr
      NAMES INTO :*output_namesbuf_ptr;
```

DESCRIBE places the descriptions of the variables to be output from the database into the location in memory pointed to by `output_sqlda_ptr`, and the names of the columns into the location pointed to by `output_namesbuf_ptr`. For code to allocate memory, see [Allocate Memory for the Values](#) on page 10-31.

## Handle Scale

If your program must handle numeric values with scale, read the scale information from the output SQLDA structure. The DESCRIBE statement places this information in bits 0 through 7 of the `data_len` field in the sqlvar array. If you ignore scale, set the `data_len` field to 0, causing data truncation. Otherwise, save the scale information and write a function to handle the scale.

## Display the Output

To display output from the database after the cursor FETCH, perform these steps:

1. Set pointers to the beginning of the first SQLVAR array and to the beginning of the names buffer.
2. Get the number of output columns from the SQLDA structure.
3. Write the column name to the output file using the names buffer pointer (only if you are doing a repetitive display of the column names).
4. Read the `data_type` field from the SQLVAR array to get the data type of the column value to be written.
5. Write the value at the location pointed to by the `var_ptr` field from the SQLVAR array. The steps to use depend on the data type of the value.

The sequence just described displays names and values repetitively. For example:

```
EMPNUM    2000
EMPNAME   JANE ROBERTS

EMPNUM    1566
```

```
EMPNAME  CATHERINE WILLIAMS
```

```
EMPNUM   1890
```

```
EMPNAME  RICHARD SMITH
```

You can also display the column names as headings (similar to SQLCI) by executing this loop for `output_num` iterations:

1. Get the length of the column name.
2. Advance to the name.
3. Display the name with some blank space.
4. 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.

You can use data type literals to decide how to display output column values.

[Example 10-6](#) displays output.

---

#### Example 10-6. Displaying Output (page 1 of 2)

```
/* Declare, open, fetch, and close the cursor. */
/* (for code, see sample program)                */
int display_result ( sqldaptr output_sqlda_ptr,
                    char *output_namesbuf_ptr )

{
    int *len_ptr;          /* Pointer to get length info      */
                          /* from the names buffer          */
    int name_len;          /* Number of bytes in column name */
    int num_entries;       /* Number of columns to be output */
    int i;                 /* loop counter                   */
    char data_array[39];   /* Buffer to contain data to be   */
                          /* displayed (null terminated)    */
    char *data_ptr;        /* Pointer to retrieved data      */
    int data_len;          /* Data buffer size               */
    char name_array[40];   /* Buffer for null terminated     */
                          /* column name, in the format     */
                          /* tablename.colname              */
                          /* ( 8 + 1 + 30 characters )     */
    char *lastchar;        /* Last character read            */
}
```

---



---

**Example 10-6. Displaying Output** (page 2 of 2)

---

```

num_entries = output_sqlda_ptr->num_entries;

for (i=0; i < num_entries; i++)
{
    /* Position output_namesbuf_ptr to the length prefix in */
    /* the names buffer, store the length in name_len, move */
    /* the pointer past the prefix and onto a name, and store */
    /* the column name in name_array. Code is the same as */
    /* that used for input parameter names (See "Getting */
    /* Parameter Values"), except that when no name is */
    /* supplied, name_array should be assigned the string */
    /* "(EXPR)". */
    /*
    /* If you want to display the column names once (as SQLCI */
    /* does), rather than repetitively with each FETCH, */
    /* display all the names at this point. The remaining */
    /* code here assumes a repetitive display of column names */
    /* and their associated values. */

    switch (output_sqlda_ptr->sqlvar[i].data_type)
    {
        case _SQLDT_ASCII_F : /* char data type */
            data_ptr = (char *)output_sqlda_ptr->sqlvar[i].var_ptr;
            data_len = output_sqlda_ptr->sqlvar[i].data_len;
            ...
            strncpy (data_array, data_ptr, data_len);
            data_array[data_len] = '\0';
            printf( "%-40s %s\n", name_array, data_array);
            ...
            break;

            ...

            /* Continue to handle all the possible data types for */
            /* output values and write the data pointed to by */
            /* the var_ptr field in the output SQLDA in a format */
            /* depending on the data type. */
            /* (For complete code, see sample program.) */

        } /* End of SWITCH statement to display values */
        /* according to data type */
    } /* End of loop to process each column */
}

```

---

# Developing a Dynamic SQL Pathway Server

Follow these guidelines to develop a C server that interfaces with Pathway and uses dynamic SQL statements. Except for constructing the SQL statement, these steps are not unique to servers using NonStop SQL/MP. You perform these steps in addition to the tasks you would perform for any dynamic SQL program.

1. Use the `#include` directive to copy the declarations in the `cextdecs` file for the `FILE_OPEN_`, `READUPDATE`, and `REPLY` procedures.
2. Define storage for the messages the server will receive from the SCREEN COBOL requester.
3. Define a character string to contain the dynamic SQL statement the program will construct from the input.
4. Call the `FILE_OPEN_` and `READUPDATE` procedures to open and read `$RECEIVE`. For information about reading `$RECEIVE`, see the *Guardian Programmer's Guide*.
5. Construct the dynamic SQL statement. Check the values passed from the requester in the buffer to determine the syntax of the statement. As you process each value, concatenate the corresponding text to form the statement.

For example, suppose that the screen describes a personnel record. If any column does not have a value, the user can enter an N. The request message you define is named `list_msg`. This example checks the `empnum` field in `list_msg` and, if required, concatenates the text "empnum" to the dynamic SQL statement:

```
char statement[200];
...
strcpy (statement, "SELECT");
if (list_msg.empnum != 'N') strcat (statement, " empnum");
...
```

The SQL statement now contains the string "SELECT empnum". You continue to construct the entire statement based on values entered by the user.

6. After you construct the statement, pad the remainder of the buffer, including the null terminator position, with blanks.
7. Compile and run the SQL statement using either the `PREPARE` and `EXECUTE` statements or the `EXECUTE IMMEDIATE` statement.
8. Construct the reply message to return information to the SCREEN COBOL requester. The first field in the reply message must contain the reply code to communicate with the SCREEN COBOL requester. The remaining fields in the message contain data returned by the SQL statement.
9. Call the `REPLY` procedure to send the reply message to the SCREEN COBOL requester.

If possible, avoid having fields in your requester or server messages that contain an odd number of bytes. There are some subtle differences in the way SCREEN COBOL and C generate fields in records when fields contain an odd number of bytes. Also, some C functions generate a null byte terminator for character strings. If your server contains a message with null terminators, the message will not match the one sent from the SCREEN COBOL requester. Therefore, to avoid these problems, follow these guidelines:

- Use DDL to describe the request and reply messages and then use the C form of the structures derived from the DDL compiler. The DDL compiler does not append a null terminator to C character strings.
- In the SCREEN COBOL requester, avoid constructing messages by listing several data items in the SEND statement. Instead, send a single structure to the C server.
- Ensure that the C server does not use logic that expects to find null terminators in the request message. For example, the `printf`, `strcpy`, and `strlen` functions expect the null byte. Consider moving the request data to another location that allows room for the null byte and processing the data from the new location.

## Dynamic SQL Sample Programs

These pages contain two complete dynamic SQL programs in C. The first program processes a SELECT statement that is partially coded into the program; the user supplies the WHERE clause. The second program allows the user to enter any SQL statement.

### Basic Dynamic SQL Program

The basic sample program contains a SELECT statement to find the average salary for a selection of rows in the employee table. The program prompts the user for the selection criteria and constructs the statement by adding a WHERE clause.

This program is an elementary one because there are no input parameters and there is only one output variable (the salary column is the only column described in the output SQLDA, and the average salary is the only value output to the user). Because no parameter names or column headings are required, names buffers are not necessary.

This program allocates memory at compile time by using INCLUDE SQLDA and specifying 1 output variable. You can specify 1 output variable because you know you are only reporting data for one column. You must still assign the memory location of the value to be output (in this case, the average) to the `var_ptr` field.

To run this program, you need a DEFINE that points to the employee table in the sample database. A complete set of DEFINES might look like this:

```
SET DEFMODE ON
ALTER DEFINE =_DEFAULTS, CATALOG \SYS1.$VOL1.TESTCAT
ALTER DEFINE =_DEFAULTS, VOLUME \SYS1.$VOL1.TESTVOL
```

```
SET DEFINE CLASS MAP  
ADD DEFINE =EMPLOYEE, FILE PERSNL.EMPLOYEE
```

Following is sample output for the program:

```
47> run ezout  
PLEASE ENTER:  
1 -- To find average salary based on employee number  
2 -- To find average salary based on job code  
3 -- To find average salary based on department number  
1  
Please enter the comparison criteria:  
(for example: > 500, = 1000, <= 250)  
> 500  
THE AVERAGE SALARY IS: 52250  
48>
```

The commented program listing appears in [Example 10-7](#) on page 10-39.

---

**Example 10-7. Basic Dynamic SQL Program (page 1 of 4)**

```

1  /* This program finds the average salary for employees      */
2  /* according to criteria established by the user. The        */
3  /* program contains a SELECT statement for the EMPLOYEE     */
4  /* table; the user enters the selection criteria, which      */
5  /* the program concatenates to the SELECT statement as a    */
6  /* WHERE clause.   */
7  /* */
8  #pragma inspect
9  #pragma symbols
10 #pragma SQL
11 #pragma xmem
12 #pragma runnable
13 #pragma nolmap
14 #pragma nomap
15
16 #pragma nolist
17 #include <stdioh>
18 #include <stdlibh>
19 #include <stringh>
20 #include <memoryh>
21
22 /* For SQL error reporting:                                */
23 #include <cextdecs (SQLCADISPLAY)>
24
25 /* For SQL data type literals:                             */
26 #include <sqlh>
27 #pragma list
28
29 #define MAXCMD 512
30
31 /* Global variables:                                       */
32 int sqlcode;          /* for error checking      */
33 long average;         /* for output value        */
34 int i;                /* loop counter            */
35 char temp[100];       /* temporary storage for user input */
36
37 /* Buffers for storing SQL statements are always blank     */
38 /* padded, never null terminated.                          */
39 char cmd[MAXCMD];     /* for SQL statement user enters */
40
41
42 /* Include SQLCA for error checking, SQLSA for dynamic SQL */
43 /* processing information:                                  */
44 exec sql INCLUDE SQLCA;
45 exec sql INCLUDE SQLSA;
46
47 /* The program will have only one output column, SALARY.   */
48 /* Since we will be generating its average, we do not need */
49 /* to print the column name--we can therefore omit          */
50 /* declaring a names buffer. We will use this SQLDA (not   */
51 /* a template) because we are not allocating memory        */
52 /* dynamically--we know we need only one output variable.  */
53
54 /* C differs from Pascal and COBOL in requiring that the    */
55 /* INCLUDE SQLDA statement appear within a DECLARE section. */
56 exec sql BEGIN DECLARE SECTION;
57     exec sql INCLUDE SQLDA (osqlda,1) ;
58 exec sql END DECLARE SECTION;
59
60 /* ----- */
61
```

---

---

**Example 10-7. Basic Dynamic SQL Program (page 2 of 4)**

---

```

62  /* Declare error handling function:                                */
63  void sql_err()
64  {
65      SQLCADISPLAY ((int *) &sqlca);
66  }
67
68  /* ----- */
69
70  /* Declare WHENEVER clause for error checking:                      */
71  exec sql WHENEVER SQLERROR CALL :sql_err;
72
73  /* ----- */
74  void blank_pad(char *buf, size_t size)
75  /*
76  /* For blank padding character strings to send to SQL              */
77  /*
78  {
79      size_t i;
80
81      i = strlen(buf);
82      if (i < size)
83          memset(&buf[i], ' ', size - i);
84  }
85
86  /* ----- */
87
88
89      EXEC SQL BEGIN DECLARE SECTION;
90  void process_and_execute ( char *cmd )
91  {
92      char (*prep_cmd)[ MAXCMD ];
93      /* SQL requires array of char, but we are passing in a          */
94      /* pointer to char. We therefore create a pointer to            */
95      /* array of char for use by SQL.                                  */
96
97      EXEC SQL END DECLARE SECTION;
98
99      blank_pad (cmd, MAXCMD);
100
101      prep_cmd = cmd;
102
103      exec sql PREPARE dyncmd FROM :*prep_cmd;
104
105      strncpy (osqla.eye_catcher,SQLDA_EYE_CATCHER, 2);
106
107      osqla.num_entries = 1;
108
109      /* Initialize ind_ptr to NULL.You must always initialize        */
110      /* this field, even when the program is not handling null        */
111      /* values.   */
112      osqla.sqlvar[0].ind_ptr = NULL;
113
114      exec sql DESCRIBE dyncmd INTO :osqla;
115
116      /* SQL tells you what it has to work with; you then              */
117      /* communicate what your variable is like to SQL; you            */
118      /* might want to look at the data_type field and adjust          */
119      /* Here, we're just putting it into a LONG and ignoring          */
120      /* scale.  */
121

```

---

**Example 10-7. Basic Dynamic SQL Program (page 3 of 4)**


---

```

122  /* set DATA_TYPE to long:                                */
123  osqlda.sqlvar[0].data_type = _SQLDT_32BIT_U;
124
125  /* set data_len to 4 bytes; leave scale as 0 in            */
126  /* upper byte of data_len:                                  */
127  osqlda.sqlvar[0].data_len = 4;
128
129  /* set VAR_PTR to point to the address of the output value:*/
130  osqlda.sqlvar[0].var_ptr = (long)&average;
131
132  exec sql BEGIN WORK;
133
134  exec sql DECLARE c1 CURSOR FOR dyncmd;
135  exec sql OPEN c1;
136  exec sql FETCH c1 USING DESCRIPTOR :osqlda;
137
138  if (sqlcode >= 0)
139      printf("\nThe average salary is: %ld\n",average);
140
141  exec sql CLOSE c1;
142
143  exec sql COMMIT WORK;
144
145  } /* end of process_and_execute */
146
147  /* ----- */
148
149  void get_cmd( char *cmd )
150  /*
151  /* Assigns a SELECT statement to the statement buffer.
152  /* Gets the WHERE clause from the user and concatenates it
153  /* to the SELECT statement.
154  /* */
155
156  {
157      char column[9];      /* column to be used in WHERE clause */
158      int sel_index;      /* selects column for WHERE clause */
159      char predicate[10]; /* comparison predicate for WHERE */
160                          /* clause */
161      size_t len;         /* for length of command, to use in */
162                          /* blanking out null terminator for */
163                          /* transmission to SQL */
164
165      strcpy (cmd, "SELECT AVG(SALARY) FROM =EMPLOYEE WHERE ");
166
167      /* Create a simple menu:
168      printf("\nPlease enter:\n\n");
169      printf("1 -- to find average salary based on employee number\n");
170      printf("2 -- to find average salary based on job code\n");
171      printf("3 -- to find average salary based on department number\n\n");
172
173      fgets(temp, (int)sizeof(temp), stdin);
174      sscanf(temp, "%d", &sel_index);
175
176      /* Initialize column and predicate to blanks:
177
178      memset(column, ' ', 9);
179
180      memset(predicate, ' ', 10);
181

```

---

**Example 10-7. Basic Dynamic SQL Program (page 4 of 4)**

```

182     switch (sel_index)
183     {
184         case 1 : strcpy(column, "EMPNUM ");
185                 break;
186         case 2 : strcpy(column, "JOBCODE ");
187                 break;
188         case 3 : strcpy(column, "DEPTNUM ");
189                 break;
190     }
191
192     printf("\nPlease enter the comparison criteria:\n");
193     printf("(for example: > 500, = 1000, <= 250)\n\n");
194
195     fgets(temp, (int)sizeof(temp), stdin);
196     sscanf(temp, "%[^\\n]", predicate);
197
198     /* Construct the SQL statement: */
199     strcat (cmd, column);
200     strcat (cmd, predicate);
201
202     /* Get length of command string and blank out null */
203     /* terminator for transmission to SQL: */
204     len = strlen(cmd);
205     cmd[len] = ' ';
206
207 } /* end of get_cmd */
208
209 /* ----- */
210
211 main()
212 {
213     /* Initialize command string to blanks: */
214     setmem (cmd, MAXCMD, ' ');
215
216     /* Get SQL statement from the user: */
217     get_cmd(cmd);
218
219     /* Compile the statement, access the SQL database, and */
220     /* report the result: */
221     process_and_execute(cmd);
222
223 } /* end of main */

```

**Detailed Dynamic SQL Program**

This program allows the user to enter any statement. The program prepares and runs the statement in a TMF transaction. The code is independent of any database because the program does not reference database definitions; only the entered statements reference a particular database.

The program performs these operations, which characterize dynamic SQL programs:

- Declares an SQLSA to determine the number of input parameters or output variables.
- Declares an SQLDA to describe input parameters and another to describe output variables. Because the program is allocating memory at run time, the SQLDA is declared as a template and allocated dynamically when the query is run.



- Defines a buffer to store output variables, with storage for column values of different data types.
- Defines a buffer to store input parameters, with storage for parameter values of different data types.
- Prepares the SQL statement and assigns it a statement name. (Note: statement and cursor host variables are not used in this program.)
- Determines the data types of the input parameters and moves them to the host variables of the corresponding data types.
- Determines the data types of the output variables and moves them to the host variables of the corresponding data types.
- Sets up the SQLDA to point to the storage for the variables referenced by the query. The storage is allocated at run time.
- Using the input SQLDA if there were parameters, either performs a cursor FETCH for a SELECT statement or runs a non-SELECT statement.

Before running the program, command interpreter ADD DEFINE commands were entered to associate tables orders and odetail with logical names =orders and =odetail, respectively. The sample query shown selects order numbers and customer numbers from the orders table where the order includes part number 6400.

Following is sample output from the program. The program prompts for input with the >> symbol. A semicolon is required to terminate input.

```

33> run cdynobj
This is DYNAMIC SQL test.
Enter SQL statement or SAME to reuse last statement or END:
>>select ordernum, custnum
from =orders where ordernum in
(select ordernum from =odetail where partnum = 6400);
ORDERS.ORDERNUM 200320
ORDERS.CUSTNUM 21
ORDERS.ORDERNUM 300350
ORDERS.CUSTNUM 543
ORDERS.ORDERNUM 800660
ORDERS.CUSTNUM 3210
ORDERS.ORDERNUM 400410
ORDERS.CUSTNUM 7654
--- 4 row(s) selected.
Enter SQL statement or SAME to reuse last statement or END:
>>end;
End of current session
34>

```

The commented program listing appears in [Example 10-8](#).

---

**Example 10-8. Detailed Dynamic SQL Program** (page 1 of 22)

---

```

1  /*****
2  /*
3  /* This program can accept any DDL or DML statement from the  */
4  /* terminal, prepare the statement, prompt for parameter      */
5  /* values, run the statement and output the result to the     */
6  /* terminal. Records returned from a SELECT operation are     */
7  /* displayed with column names                               */
8  /*
9  /*****
10
11  #pragma inspect
12  #pragma symbols
13  #pragma SQL
14  #pragma xmem
15  #pragma runnable
16
17  #pragma nolist
18  #include <stdio.h>
19  #include <stdlib.h>
20  #include <string.h>
21  #include <memory.h>
22  #include <cextdecs (SQLCADISPLAY)>
23  #pragma list
24
25  #include <sqlh>
26
27  /*****
28  /* Declare Section -- for host variable declarations          */
29  /*****
30
31  exec sql begin declare section;
32
33  int sqlcode;                      /* sqlcode (required)    */
34  exec sql include sqlca;
35  exec sql include sqlsa;
36
37  /* ----- */
38  /* Include sqllda to get SQLDA_TYPE struct and SQLVAR_TYPE   */
39  /* struct declarations   */
40  /* ----- */
41  /* Note for SQLDA structure template:                          */
42  /* ----- */
43  /*
44  /* The template for sqllda struct (SQLDA_TYPE) is declared to */
45  /* contain 1 sqlvar entry (SQLVAR_TYPE). This is done to get  */
46  /* easy addressability to the sqlvars array. When allocating  */
47  /* memory for the sqllda and the sqlvars entries, allocate    */
48  /* memory for:   */
49  /*
50  /* sizeof( struct SQLDA_TYPE ) +
51  /* (num_sqllda_entries - 1) * sizeof( struct SQLVAR_TYPE )
52  /*
53  /* ----- */
54  exec sql include sqllda (dummy_da, 1, dummy_names, 1);
55
56  typedef struct SQLDA_TYPE *sqldaptr;
57

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 2 of 22)**


---

```

58      /* ----- */
59      /* SQLDAs and names buffers for input and output variables */
60      /* ----- */
61      sqldaptr sda_i;          /* ptr to input sqlda */
62      sqldaptr sda_o;          /* ptr to output sqlda */
63
64      /* To give SQL reasonable size information for the names */
65      /* buffers, pointers to arrays of 1000 chars are */
66      /* currently used. The program will still allocate */
67      /* memory just for the required size for the names buffer; */
68      /* but such a reference in the embedded SQL statements */
69      /* lets SQL get more reasonable sized data (other than */
70      /* 1 if a char pointer is used). If enough memory, */
71      /* as reported in the SQLSA after the PREPARE statement, */
72      /* is allocated for the names buffer, SQL will not use */
73      /* (hence, will overwrite) any undesired memory locations. */
74
75      typedef char (*arrayptr) [1000];
76      arrayptr cname_i;        /* ptr to input names buffer */
77      arrayptr cname_o;        /* ptr to output names buffer */
78
79      /* ----- */
80      /* Buffers for storing SQL statements are always blank padded, */
81      /* never null terminated */
82      /* ----- */
83      #define max_query_size 512
84      char host1[max_query_size + 1]; /* accepts SQL string */
85      char host2[max_query_size + 1]; /* copy of the last SQL stmt */
86
87      exec sql end declare section;
88
89      /* ----- */
90      /* The following UNION is defined for pointers to buffers of */
91      /* different (SQL) data types. This program does not handle */
92      /* FLOAT, DOUBLE PRECISION, or DATETIME */
93      /* ----- */
94      union in_out_ptrs_u {
95          char      *char_ptr;          /* for CHAR/VARCHAR */
96          short     *smallint_ptr;      /* SMALLINT */
97          unsigned short *usmallint_ptr; /* UNSIGNED SMALLINT */
98          long      *integer_ptr;       /* INTEGER */
99          unsigned long *uinteger_ptr;  /* UNSIGNED INTEGER */
100         /* long long *longint_ptr;    64-BIT INTEGER */
101     } in_out_ptrs;
102
103     static short last_query_size = 0; /* num bytes in last query */
104     char datatype_name[50];          /* to display datatype name */
105
106     /* ----- */
107     /* Terminator character when requesting user query (semicolon) */
108     /* ----- */
109     #define QUERY_TERMINATOR (char) ';'
110     /* Cast as char because C treats character constants as type int, */
111     /* and we want to reference it as type char in the function */
112     /* prototype */
113
114     /* ----- */
115     /* Terminator character when requesting input param values (EOL) */
116     /* ----- */
117     #define PARAM_TERMINATOR (char) '\n'
118

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 3 of 22)**

```

119  extern sqldaptr allocate_sqlda ( int num_entries );
120  extern short   get_string ( char *data_array,
121                          short array_size,
122                          short nullit,
123                          char terminator );
124  extern char     *get_dtname ( short datatype );
125
126  /* ***** */
127  /* FUNCTION display_result */
128  /* This function accepts the output sqlda and the */
129  /* output names buffer as parameters and displays the */
130  /* output of a select statement with the following */
131  /* format: */
132  /* */
133  /* tablename.colname <data retrieved> */
134  /* tablename.colname <data retrieved> */
135  /* */
136  /* The display is currently restricted to at most */
137  /* 38 characters; this restriction can be easily */
138  /* relaxed by wrapping the display lines */
139  /* */
140  /* Return: 0 if successful */
141  /* -1 if failure */
142  /* */
143  /* ***** */
144
145  int display_result ( sqldaptr sqlda, /* ptr to output sqlda */
146                    char *nb )      /* ptr to names buffer */
147  { /* begin display_result */
148
149      short *len_ptr;          /* int ptr to get the length */
150                              /* from the names buffer */
151      short name_len;          /* num bytes in a name */
152      short num_entries;       /* number of sqlvar entries */
153      short i;                /* loop index */
154      char data_array[39];     /* buffer to contain data to */
155                              /* be displayed (null termi- */
156                              /* nated) */
157      char *data_ptr;          /* ptr to retrieved data */
158      short data_len;          /* data buffer size */
159      char name_array[40];     /* buffer to contain null */
160                              /* terminated name in a */
161                              /* <tablename>.<colname> format */
162                              /* [ 8 + 1 + 30 chars ] */
163      char *lastchar;
164
165      num_entries = sqlda->num_entries;
166
167      for (i=0; i < num_entries; i++)
168      {
169          len_ptr = (short *) nb; /* get to length prefix */
170          name_len = *len_ptr;
171          nb += 2;                /* advance nb to skip the */
172                              /* 2-byte length prefix */
173
174          /* get null terminated name in name_array */
175          if (name_len == 0)
176              strcpy(name_array, "(EXPR)"); /* default name */
177          else
178

```

**Example 10-8. Detailed Dynamic SQL Program (page 4 of 22)**

```

179     {
180         lastchar = nb + (name_len - 1);
181         if ( *lastchar == ' ' ) /* last character is blank */
182                                 /* that SQL inserts to make */
183                                 /* length info fall on an */
184                                 /* even byte boundary */
185                                 /* (the name had an odd */
186                                 /* number of characters) */
187
188         { strncpy( name_array, nb, name_len - 1);
189           name_array[name_len - 1] = '\0';
190         }
191         else
192         { strncpy( name_array, nb, name_len);
193           name_array[name_len] = '\0';
194         }
195     }
196
197     /* advance nb to the next name */
198     nb = lastchar + 1;
199
200     /* ----- */
201     /* Display data depending on data type */
202     /* ----- */
203
204     switch (sqlda->sqlvar[i].data_type) {
205     /* ----- */
206     case _SQLDT_ASCII_F :          /* CHAR data type */
207
208         data_ptr = (char *) sqlda->sqlvar[i].var_ptr;
209         data_len = sqlda->sqlvar[i].data_len;
210
211         if (data_len <= 38)
212         { strncpy( data_array, data_ptr, data_len );
213           data_array[data_len] = '\0';
214           printf( "%-40s %s\n", name_array, data_array );
215           fflush (stdout);
216         }
217         else
218         {
219             /* display first 38 characters of data */
220             printf( "%-40s %.38s\n", name_array, data_ptr );
221             fflush (stdout);
222         }
223         break;
224
225     /* ----- */
226     case _SQLDT_ASCII_V :          /* VARCHAR datatype */
227
228         data_ptr = (char *) sqlda->sqlvar[i].var_ptr;
229         len_ptr = (short *) data_ptr; /* length prefix */
230         data_ptr += 2;               /* skip length prefix */
231
232         if ( *len_ptr <= 38)
233         { if ( *len_ptr != 0 )      /* filter zero length */
234           { strncpy( data_array, data_ptr, *len_ptr );
235             data_array[ *len_ptr ] = '\0';
236             printf( "%-40s %s\n", name_array, data_array );
237             fflush (stdout);
238           }
239         }
240         else
241         {

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 5 of 22)**

---

```

241         /* display first 38 characters of data */
242         printf( "%-40s %.38s\n", name_array, data_ptr );
243         fflush (stdout);
244     }
245
246     break;
247
248     /* ----- */
249     case _SQLDT_16BIT_S :           /* 16bit numeric */
250
251         in_out_ptrs.smallint_ptr = (short *) sqllda->sqlvar[i].var_ptr;
252         printf( "%-40s %hd\n", name_array, *in_out_ptrs.smallint_ptr);
253         fflush (stdout);
254         break;
255
256     /* ----- */
257     case _SQLDT_16BIT_U :           /* 16 bit unsigned numeric */
258
259         in_out_ptrs.usmallint_ptr =
260             (unsigned short *) sqllda->sqlvar[i].var_ptr;
261         printf( "%-40s %hu\n", name_array, *in_out_ptrs.usmallint_ptr);
262         fflush (stdout);
263         break;
264
265     /* ----- */
266     case _SQLDT_32BIT_S :           /* 32 bit signed numeric */
267
268         in_out_ptrs.integer_ptr = (long *) sqllda->sqlvar[i].var_ptr;
269         printf( "%-40s %ld\n", name_array, *in_out_ptrs.integer_ptr);
270         fflush (stdout);
271         break;
272
273     /* ----- */
274     case _SQLDT_32BIT_U :           /* 32 bit unsigned numeric */
275
276         in_out_ptrs.uinteger_ptr =
277             (unsigned long *) sqllda->sqlvar[i].var_ptr;
278         printf( "%-40s %lu\n", name_array, *in_out_ptrs.uinteger_ptr);
279         fflush (stdout);
280         break;
281
282     /* ----- */
283     default:                       /* unsupported datatype */
284         printf( "***** Error for %-40s: %s Datatype is unsupported.\n",
285             name_array, get_dtname( sqllda->sqlvar[i].data_type ));
286         fflush (stdout);
287         break;
288
289     /* ----- */
290 } /* end: switch stmt */
291 } /* end: for loop */
292
293 /* place a space line */
294 printf("\n"); fflush( stdout );
295
296 return (0);
297
298 } /* end: display_result */
299

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 6 of 22)**

```

300  /* ***** */
301  /* FUNCTION request_invars */
302  /* This function accepts the input sqllda and the */
303  /* input names buffer as parameters and requests the */
304  /* input values for the needed input parameters */
305  /* */
306  /* Return: 0 if successful */
307  /* -1 if failure */
308  /* */
309  /* ***** */
310
311  int request_invars ( sqldaptr sqllda, /* ptr to input sqllda */
312                      char *nb )      /* ptr to names buffer */
313
314  { /* begin request_invars */
315
316      short *len_ptr;          /* int ptr to get the length */
317                              /* from the names buffer */
318                              /* and write len prefix to */
319                              /* varchar data buffers */
320      short name_len;          /* num bytes in a name */
321      short num_entries;       /* number of sqlvar entries */
322      short i;                /* loop index */
323
324      #define data_array_size 21
325      char data_array[data_array_size];
326                              /* buffer to get numeric data */
327                              /* max 19 digits + sign byte */
328                              /* + null terminator */
329      short data_len;          /* #bytes of input data needed */
330      short data_read;         /* #bytes of input read */
331      char name_array[31];     /* buffer to contain null */
332                              /* terminated name of the */
333                              /* input param (without the */
334                              /* leading '?' ) */
335      char *lastchar;
336      char *dummy;
337
338      num_entries = sqllda->num_entries;
339      printf( "\nPlease provide data for input params \n");
340      printf( "----- \n\n");
341      fflush( stdout);
342
343      for (i=0; i < num_entries; i++)
344      {
345          len_ptr = (short *) nb; /* get to length prefix */
346          name_len = *len_ptr;
347          nb += 2;                /* advance nb to skip the */
348                              /* 2-byte length prefix */
349
350          /* sanity check */
351          if (name_len > 30)
352          { printf("**** Error: Param name is too long. Try again.\n");
353            fflush( stdout );
354            return (-1);
355          }
356
357          /* get null terminated param name in name_array */
358          if (name_len == 0)
359              name_array[0] = '\0'; /* unnamed param */
360          else
361          {

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 7 of 22)**

---

```

362     lastchar = nb + (name_len - 1);
363     if ( *lastchar == ' ')
364     { strncpy( name_array, nb, name_len - 1);
365       name_array[name_len - 1] = '\0';
366     }
367     else
368     { strncpy( name_array, nb, name_len);
369       name_array[name_len] = '\0';
370     }
371     /* advance nb to the next name */
372     nb = lastchar + 1;
373 }
374
375
376 /* ----- */
377 /* Request input data depending on data type */
378 /* ----- */
379
380 switch (sqlda->sqlvar[i].data_type) {
381 /* ----- */
382 case _SQLDT_ASCII_F : /* CHAR data type */
383
384     in_out_ptrs.char_ptr = (char *) sqlda->sqlvar[i].var_ptr;
385     data_len = sqlda->sqlvar[i].data_len;
386
387     if (name_len > 0)
388         printf("Please enter max %d characters for ?%s: ",
389               data_len, name_array);
390     else
391         printf("Please enter max %d characters: ",
392               data_len);
393
394     if ( get_string( in_out_ptrs.char_ptr, data_len,
395                     0, PARAM_TERMINATOR ) < 0 )
396     { /* input info too long */
397       printf("\n**** Error: Input data is too long.\n");
398       fflush(stdout);
399       return (-1);
400     }
401
402     break;
403
404 /* ----- */
405 case _SQLDT_ASCII_V : /* VARCHAR data type */
406
407     in_out_ptrs.char_ptr = (char *) (sqlda->sqlvar[i].var_ptr + 2);
408     data_len = sqlda->sqlvar[i].data_len;
409
410     if (name_len > 0)
411         printf("Please enter max %d characters for ?%s: ",
412               data_len, name_array);
413     else
414         printf("Please enter max %d characters: ",
415               data_len);
416
417     if ( ( data_read = get_string( in_out_ptrs.char_ptr, data_len,
418                                   0, PARAM_TERMINATOR ) ) < 0 )
419     { /* input info too long; or some problem */
420       printf("\n**** Error: Input data is too long.\n");
421       fflush(stdout);
422       return (-1);
423     }
424

```

---



---

**Example 10-8. Detailed Dynamic SQL Program (page 8 of 22)**

---

```

425     len_ptr = (short *) sqlda->sqlvar[i].var_ptr;
426     *len_ptr = data_read;          /* #chars for the varchar buffer */
427
428     break;
429
430     /* ----- */
431     case _SQLDT_16BIT_S :      /* 16 bit signed numeric */
432     case _SQLDT_16BIT_U :      /* 16 bit unsigned numeric */
433     case _SQLDT_32BIT_S :      /* 32 bit signed numeric */
434     case _SQLDT_32BIT_U :      /* 16 bit unsigned numeric */
435
436     if (name_len > 0)
437     printf("Please enter numeric value for %s: ", name_array);
438     else
439     printf("Please enter a numeric value: ");
440
441     if ( get_string( data_array, data_array_size,
442     1, PARAM_TERMINATOR ) < 0 )
443     { /* input info too long; or some problem */
444     printf("\n*** Error: Input number is too big.\n");
445     fflush(stdout);
446     return (-1);
447     }
448
449     /* Convert input number to appropriate numeric form. */
450
451     switch (sqlda->sqlvar[i].data_type) {
452     /* ----- */
453     case _SQLDT_16BIT_S :      /* 16 bit signed numeric */
454     in_out_ptrs.smallint_ptr =
455     (short *) sqlda->sqlvar[i].var_ptr;
456     *in_out_ptrs.smallint_ptr = atoi( data_array );
457     break;
458
459     /* ----- */
460     case _SQLDT_16BIT_U :      /* 16 bit unsigned numeric */
461     in_out_ptrs.usmallint_ptr =
462     (unsigned short *) sqlda->sqlvar[i].var_ptr;
463     *in_out_ptrs.usmallint_ptr =
464     (unsigned short) atol( data_array );
465     break;
466
467     /* ----- */
468     case _SQLDT_32BIT_S :      /* 32 bit signed numeric */
469     in_out_ptrs.integer_ptr =
470     (long *) sqlda->sqlvar[i].var_ptr;
471     *in_out_ptrs.integer_ptr = atol( data_array );
472     break;
473
474     /* ----- */
475     case _SQLDT_32BIT_U :      /* 32 bit unsigned numeric */
476     in_out_ptrs.uinteger_ptr =
477     (unsigned long *) sqlda->sqlvar[i].var_ptr;
478     dummy = NULL;
479     *in_out_ptrs.uinteger_ptr = strtoul (data_array, &dummy, 10);
480
481     break;
482
483     /* ----- */
484     } /* end: inner switch */
485
486     break;

```

---

**Example 10-8. Detailed Dynamic SQL Program** (page 9 of 22)

```

487
488      /* ----- */
489      default: /* unsupported datatype */
490      printf( "**** Error for %-40s: %s Datatype is unsupported.\n",
491             name_array, get_dtname( sqlda->sqlvar[i].data_type ) );
492      fflush( stdout );
493
494      break;
495
496      /* ----- */
497  } /* end: switch stmt */
498  } /* end: for loop */
499
500  printf("\n"); fflush( stdout );
501  return (0);
502
503  } /* end: request_invars */
504
505  /* ***** */
506  /* FUNCTION get_dtname */
507  /* This function places the name of a given data type into */
508  /* the array datatype_name */
509  /* */
510  /* Return: pointer to array datatype_name */
511  /* (array is null terminated) */
512  /* ***** */
513  char *get_dtname ( short datatype )
514  { /* begin get_dtname */
515
516  switch (datatype) {
517      /* ----- */
518      case _SQLDT_ASCII_F : /* CHAR data type */
519
520          strcpy( datatype_name, "CHARACTER" );
521          break;
522
523      /* ----- */
524      case _SQLDT_ASCII_V : /* VARCHAR data type */
525
526          strcpy( datatype_name, "VARCHAR" );
527          break;
528
529      /* ----- */
530      case _SQLDT_16BIT_S : /* 16 bit signed binary */
531
532          strcpy( datatype_name, "SIGNED 16BIT NUMERIC" );
533          break;
534
535      /* ----- */
536      case _SQLDT_16BIT_U : /* 16 bit unsigned binary */
537
538          strcpy( datatype_name, "UNSIGNED 16BIT NUMERIC" );
539          break;
540
541      /* ----- */
542      case _SQLDT_32BIT_S : /* 32 bit signed binary */
543
544          strcpy( datatype_name, "SIGNED 32BIT NUMERIC" );
545          break;
546
547      /* ----- */
548      case _SQLDT_32BIT_U : /* 32 bit unsigned binary */

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 10 of 22)**

---

```

550
551     strcpy( datatype_name, "UNSIGNED 32BIT NUMERIC" );
552     break;
553
554     /* ----- */
555     case _SQLDT_64BIT_S :           /* 64 bit signed binary */
556
557         strcpy( datatype_name, "SIGNED 64BIT NUMERIC" );
558         break;
559
560     /* ----- */
561     case _SQLDT_DEC_U :           /* DECIMAL datatype: unsi */
562
563         strcpy( datatype_name, "UNSIGNED DECIMAL" );
564         break;
565
566     /* ----- */
567     case _SQLDT_DEC_LSS :        /* DECIMAL datatype: LSS */
568
569         strcpy( datatype_name, "LEADING SIGN SEPARATE DECIMAL" );
570         break;
571
572     /* ----- */
573     case _SQLDT_DEC_LSE :        /* DECIMAL datatype: LSE */
574
575         strcpy( datatype_name, "LEADING SIGN EMBEDDED DECIMAL" );
576         break;
577
578     /* ----- */
579     case _SQLDT_DEC_TSS :        /* DECIMAL datatype: TSS */
580
581         strcpy( datatype_name, "TRAILING SIGN SEPARATE DECIMAL" );
582         break;
583
584     /* ----- */
585     case _SQLDT_DEC_TSE :        /* DECIMAL datatype: TSE */
586
587         strcpy( datatype_name, "TRAILING SIGN EMBEDDED DECIMAL" );
588         break;
589
590     /* ----- */
591     default:
592
593         strcpy( datatype_name, "UNEXPECTED" );
594         break;
595
596     /* ----- */
597     } /* end: switch */
598
599     return (datatype_name);
600
601 } /* end get_dtname */
602

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 11 of 22)**


---

```

603  /* ***** */
604  /* FUNCTION get_string */
605  /* This function reads from the standard input, a character */
606  /* string, into the data_array */
607  /* The data_array will be null terminated, or blank padded, */
608  /* as requested. The reading stops when 'terminator' char is */
609  /* read or if array_size number of characters have been read, */
610  /* whichever comes first */
611  /* */
612  /* For multi-line input, all white space characters are */
613  /* replaced by blanks */
614  /* */
615  /* Return: +ve integer, if successful; = the number of chars */
616  /* read from the input (minus the terminator char) */
617  /* -1 if entered data is too long for data_array */
618  /* (if no room for the null terminator (if reqsted) */
619  /* (if array_size not enough for entire input, ie */
620  /* until the semicolon ) */
621  /* ***** */
622
623  short get_string( char *data_array, /* array to read data into */
624                  short array_size, /* max #bytes in array */
625                  short nullit,     /* if != 0, terminate */
626                                  /* data_array on return, */
627                                  /* else blank pad array */
628                  char terminator ) /* terminator character */
629
630  { /* begin get_string */
631
632  char    c;
633  short   ix;
634
635  short   i;
636
637  /* sanity check */
638  if (array_size == 0) /* buffer no good */
639      return (-1);
640
641  ix = 0;
642  while ( (c = getchar()) != terminator )
643  {
644      if ( c == '\t' || c == '\n' )
645          *(data_array + ix) = ' '; /* replace by blank */
646      else
647          *(data_array + ix) = c;
648
649      if (++ix >= array_size) /* no more room in array */
650      { if (nullit == 0) /* blankpadding requested */
651          { if ( (c = getchar()) == terminator )
652              { /* the next char was the terminator anyway. */
653                  /* just made it. also consume extra input */
654                  /* at while-loop exit */
655                  break;
656              }
657              else
658              { while ( (c = getchar()) != '\n' )
659                  { /* consume the input */ }
660                  return (-1); /* array too small */
661              }
662          }
663      else
664          /* null termination reqsted */

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 12 of 22)**

```

664     { while ( (c = getchar()) != '\n' )
665         { /* consume the input */ }
666         return (-1); /* array too small */
667     }
668 }
669 } /* end: while loop */
670
671 /* out of while loop only at terminator char. */
672 /* consume the remainder of input line */
673 if (terminator != '\n')
674     while ( (c = getchar()) != '\n' )
675         { /* consume the input */ }
676
677 /* ix points to next available slot */
678 /* null terminate or blank pad, as requested */
679 if (nullit == 0)
680     { for (i = ix; i < array_size; i++)
681         *(data_array + i) = ' '; /* blank pad */
682     }
683 else
684     *(data_array + ix) = '\0';
685
686 return (ix);
687
688 } /* end: get_string */
689
690 /* ***** */
691 /* FUNCTION read_query */
692 /* This function reads from the standard input (terminal) */
693 /* the SQL query. A semicolon marks the end of the query. */
694 /*
695 /* If the user types in END/end/E/e then the session is
696 /* stopped. If the user types in SAME/same then the last
697 /* user query is run. If the user types in an SQL
698 /* query, the query is read in 'host1' array and a copy
699 /* of it is made in 'host2' array
700 /*
701 /* Return: 0 if query read in or SAME case
702 /* -1 if END case
703 /* ***** */
704 int read_query ( void )
705 {
706     short query_len; /* length of query in bytes */
707
708     try_again:
709     printf ("\nEnter SQL statement or SAME to reuse last statement or
END:\n");
710     fflush (stdout);
711     printf (">> ");
712
713     if ( (query_len = get_string (host1, max_query_size,
714                                0, QUERY_TERMINATOR)) < 0 )
715         { printf("**** Error: Input query is too long.\n");
716           fflush( stdout );
717           goto try_again;
718         }
719
720     if ( ( strcmp(host1, "E", 1) == 0 ) ||
721         ( strcmp(host1, "e", 1) == 0 ) )
722         return (-1);
723

```

**Example 10-8. Detailed Dynamic SQL Program (page 13 of 22)**

```

724  if ( ( strcmp(host1, "same", 4) == 0 ) ||
725        ( strcmp(host1, "SAME", 4) == 0 ) )
726  { /* restore the saved query to host1 and display it */
727    strncpy ( host1, host2, max_query_size );      /* do an 'fc' */
728    printf( "\nRe-executing Query >> ");
729    host1[last_query_size] = '\0';      /* temporarily null terminate */
730    puts( host1 ); fflush (stdout);      /* display query */
731    host1[last_query_size] = ' ';      /* restore the blank */
732  }
733  }
734  else
735  { /* backup the query and remember its size */
736    strncpy ( host2, host1, max_query_size );      /* backup the query */
737    last_query_size = query_len;      /* remember size */
738  }
739  return (0);
740  } /* end: read_query */
741
742  /* ***** */
743  /* FUNCTION adjust_sqlda_scale_types */
744  /* This function takes an SQLDA as a parameter and, */
745  /* for sqlda.num_entries, adjusts the recommended */
746  /* (by SQL) data types and scales to what C supports. */
747  /* */
748  /* Setting up buffers for supported data types */
749  /* involves modifying the data_len and data_type */
750  /* of the SQLVAR entry to reflect the data attributes */
751  /* of the allocated buffers. For example, an input */
752  /* parameter or output variable with */
753  /* data_type == _SQLDT_DEC_LSS and */
754  /* data_len == 7 (assuming scale = 0) */
755  /* can be modified to have */
756  /* data_type == _SQLDT_32BIT_S and */
757  /* data_len == 4 */
758  /* and a 4 byte buffer can be allocated for it */
759  /* */
760  /* Scale is set to 0 */
761  /* Data_type is set to nearest equivalent supported */
762  /* type */
763  /* */
764  /* ***** */
765  int adjust_sqlda_scale_types ( sqldaptr sqlda )
766  { /* begin adjust_sqlda_scale_types */
767
768    int num_entries;      /* number of sqlvar entries */
769    int i;      /* loop index */
770
771    num_entries = sqlda->num_entries;
772
773    for (i = 0; i < num_entries; i++)
774    {
775      switch (sqlda->sqlvar[i].data_type) {
776        /* ----- */
777        case _SQLDT_16BIT_S :      /* SMALLINT */
778        case _SQLDT_16BIT_U :      /* UNSIGNED SMALLINT */
779        case _SQLDT_32BIT_S :      /* INTEGER */
780        case _SQLDT_32BIT_U :      /* UNSIGNED INTEGER */
781        case _SQLDT_64BIT_S :      /* SIGNED LARGEINT */
782
783          /*-----*/
784          /* set scale information to 0 */
785          /*-----*/

```

**Example 10-8. Detailed Dynamic SQL Program (page 14 of 22)**

```

787         sqlda->sqlvar[i].data_len = sqlda->sqlvar[i].data_len & 0377;
788
789
790         break;
791
792         /* ----- */
793         /* DECIMAL is supported; if your database has DECIMAL */
794         /* items, you might not want to translate to 32-bit */
795         /* integers as this program does */
796         case _SQLDT_DEC_U :                /* DECIMAL unsigned */
797
798         /* The following types are unsupported: */
799         case _SQLDT_DEC_LSS :                /* DECIMAL LSS */
800         case _SQLDT_DEC_LSE :                /* DECIMAL LSE */
801         case _SQLDT_DEC_TSS :                /* DECIMAL TSS */
802         case _SQLDT_DEC_TSE :                /* DECIMAL TSE */
803
804         /*----- */
805         /* Map to _SQLDT_32BIT_S type */
806         /* Length info must be set to 4 bytes for */
807         /* scale information to be set to 0 */
808         /* Note: for DECIMAL, you might want to save */
809         /* the scale information instead of setting */
810         /* to zero as this program does */
811         /*----- */
812         sqlda->sqlvar[i].data_type = _SQLDT_32BIT_S;
813         sqlda->sqlvar[i].data_len = 4;        /* and scale is 0 */
814
815         break;
816
817         /* ----- */
818         default: /* UNSUPPORTED types or do not need adjustments */
819
820         break;                /* (Nothing to be done) */
821
822         /* ----- */
823     } /* switch stmt */
824
825 } /* for loop */
826
827 return (0);
828
829 } /* end adjust_sqlda_scale_types
831
832 /* ***** */
833 /* FUNCTION setupvarbuffers */
834 /* This function takes an SQLDA as a parameter and, */
835 /* for sqlda.num_entries, allocates the data buffers */
836 /* for appropriate lengths. For each sqlvar, */
837 /* sqlda.sqlvar[i].var_ptr is set to point to that */
838 /* buffer */
839 /* */
840 /* The sqlda is also changed by altering unsupported */
841 /* data types to the nearest equivalent data types */
842 /* and by setting scale information to 0 */
843 /* */
844 /* sqlda.num_entries is assumed to have a valid value. */
845 /* */
846 /* Return: 0 if successful */
847 /* -1 if failure */
848 /* */
849

```

**Example 10-8. Detailed Dynamic SQL Program (page 15 of 22)**

```

850  int setupvarbuffers ( sqldaptr sqlda )
851  {
852      /* begin setupvarbuffers */
853      int num_entries;          /* number of sqlvar entries */
854      int mem_reqd;             /* buffer size */
855      int i;                    /* loop index */
856
857      /* ----- */
858      /* Handle unsupported types; set scale information to 0. */
859      /* ----- */
860      adjust_sqlda_scale_types( sqlda );
861
862      num_entries = sqlda->num_entries;
863      for (i = 0; i < num_entries; i++)
864      {
865          switch (sqlda->sqlvar[i].data_type) {
866              /* ----- */
867              case _SQLDT_ASCII_F : /* CHAR datatype */
868                  mem_reqd = sqlda->sqlvar[i].data_len;
869                  break;
870
871              /* ----- */
872              case _SQLDT_ASCII_V : /* VARCHAR datatype */
873                  mem_reqd = sqlda->sqlvar[i].data_len + 2;
874                  break;
875
876              /* ----- */
877              case _SQLDT_16BIT_S : /* SMALLINT */
878              case _SQLDT_16BIT_U : /* UNSIGNED SMALLINT */
879              case _SQLDT_32BIT_S : /* INTEGER */
880              case _SQLDT_32BIT_U : /* UNSIGNED INTEGER */
881
882                  /*----- */
883                  /* NOTE ON SCALE INFORMATION */
884                  /*----- */
885                  /* Bits 0 through 7 of sqlda->sqlvar[i].data_len */
886                  /* have the scale information for the numeric */
887                  /* data types. Either remember this scale */
888                  /* information and later use the values in the */
889                  /* host variables appropriately or set the */
890                  /* scale information to 0 (which can lead to */
891                  /* truncated values on retrievals and inability */
892                  /* to provide scaled values through input */
893                  /* parameters) */
894                  /*----- */
895                  /* Set scale information to 0 (see note above) */
896                  /*----- */
897                  sqlda->sqlvar[i].data_len = sqlda->sqlvar[i].data_len & 0377;
898
899                  /*----- */
900                  /* Extract length from bits 8:15 */
901                  /*----- */
902                  mem_reqd = sqlda->sqlvar[i].data_len & 0377;
903                  break;
904
905              /* ----- */
906              default:
907                  /* UNSUPPORTED types */
908
909                  printf( "\n**** Error: Unsupported Datatype: %s\n",
910                          get_dtname( sqlda->sqlvar[i].data_type ));
911                  return (-1);
912

```



**Example 10-8. Detailed Dynamic SQL Program (page 16 of 22)**


---

```

913      /* ----- */
914      }          /* switch statement */
915
916      /* ----- */
917      /* Allocate memory for the data buffer and assign */
918      /* byte address of the data buffer to var_ptr of */
919      /* sqlvar[i]: */
920      /* ----- */
921      sqlda->sqlvar[i].var_ptr = (long) (malloc (mem_reqd));
922
923      }          /* for loop */
924
925      return (0);      /* successful buffer allocation */
926      }          /* end: setupvarbuffers */
927
928      /* ***** */
929      /* FUNCTION allocate_sqlda */
930      /* This function allocates (using malloc): */
931      /* an sqlda structure with 'num_entries' entries; */
932      /* the function also initializes the sqlda and sqlvars. */
933      /* */
934      /* Return codes: sqlda pointer if successful */
935      /* NULL if failure */
936      /* ***** */
937
938      sqldaptr allocate_sqlda ( int num_entries )
939                          /* number of sqlvar_s entries */
940
941      { /* begin allocate_sqlda */
942
943      /* local variables */
944      sqldaptr sqlda;          /* pointer to be returned*/
945      int mem_reqd;            /* num bytes required to */
946                                /* allocate sqlda */
947      short i;                /* loop index */
948
949      sqlda = NULL;            /* init pointer */
950
951      /* return NULL if 0 entries requested */
952      if (num_entries == 0)
953      return (sqlda);
954
955      /* allocate sqlda */
956      mem_reqd = sizeof( struct SQLDA_TYPE ) +
957                  ((num_entries - 1) * sizeof( struct SQLVAR_TYPE ));
958      if ( (sqlda = (sqldaptr) malloc (mem_reqd)) == NULL )
959                                /* memory allocation failed */
960      return (sqlda);          /* return error condition */
961
962      /* Initialize sqlda; constant sqlda_eye_catcher is defined */
963      /* by the C compiler and is always 2 characters: */
964      strncpy( sqlda -> eye_catcher, SQLDA_EYE_CATCHER, 2);
965
966      sqlda -> num_entries = num_entries;
967
968      /* Initialize ind_ptr to NULL. ind_ptr must always be */
969      /* initialized, even when the program does not handle null */
970      /* values */
971      for (i=0; i < num_entries; i++)
972      sqlda -> sqlvar[i].ind_ptr = NULL;
973
974      return (sqlda);          /* successful allocation and init. */
975      } /* end allocate_sqlda */

```

---

---

**Example 10-8. Detailed Dynamic SQL Program (page 17 of 22)**

---

```

976
977  /* ***** */
978  /* FUNCTION free_sqlda */
979  /* This function accepts an sqlda as a parameter and */
980  /* frees all memory that was allocated for the data */
981  /* buffers (pointed to as sqlvar[i].var_ptr) */
982  /* and for the sqlda and sqlvar entries */
983  /* */
984  /* The function assumes that if a valid sqlda is */
985  /* passed, then sqlda.num_entries has a valid value */
986  /* */
987  /* ***** */
988  int free_sqlda ( sqldaptr sqlda)
989  { /* begin free_sqlda */
990  int num_entries;          /* number of sqlvar entries */
991  short i;                  /* loop index */
992  char *buf_ptr;            /* pointer to sqlvar buffer */
993
994  /* sanity check */
995  if (sqlda == NULL)
996      return (0);
997
998  num_entries = sqlda->num_entries;
999  for (i = 0; i < num_entries; i++)
1000      { if ( (buf_ptr = (char *) sqlda->sqlvar[i].var_ptr) != NULL)
1001          free( buf_ptr );
1002      }
1003
1004  free ( (char *) sqlda );          /* freeup the sqlda memory */
1005
1006  return (0);
1007
1008  } /* end free_sqlda */
1009
1010  /* ***** */
1011  /* FUNCTION cleanup */
1012  /* This function frees up the allocated memory for the */
1013  /* input and output sqldas and names buffers and the */
1014  /* data buffers allocated for the sqldas */
1015  /* ***** */
1016  void cleanup ()
1017  { /* cleanup */
1018
1019      free_sqlda( sda_i );          /* free input sqlda. */
1020      free_sqlda( sda_o );          /* free output sqlda. */
1021      sda_i = sda_o = NULL;         /* init pointers */
1022
1023      if (cname_i != NULL)
1024          free ( (char *) cname_i ); /* free i/p names buffer */
1025      if (cname_o != NULL)
1026          free ( (char *) cname_o ); /* free o/p names buffer */
1027
1028      cname_i = cname_o = NULL; /* init pointers */
1029
1030  } /* cleanup */
1031
1032  main ()
1033  {

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 18 of 22)**

```

1034 /* ----- */
1035 /* local variables */
1036 /* ----- */
1037 int out_numvars; /* number of output variables */
1038 int in_numvars; /* number of input variables */
1039 unsigned int out_nameslen; /* size of o/p names buffer */
1040 unsigned int in_nameslen; /* size of i/p names buffer */
1041 int status;
1042 unsigned long num_fetches; /* #records fetched */
1043
1044 /* init pointers */
1045 sda_i = sda_o = NULL; /* sqllda pointers */
1046 cname_i = cname_o = NULL; /* names buffer pointers */
1047
1048 /* blank extra byte in host1, host2 */
1049 host1[ max_query_size ] = host2[ max_query_size ] = ' ';
1050
1051 printf("This is DYNAMIC SQL test.\n");
1052 fflush (stdout);
1053
1054 /*****
1055 /* Input SQL query from terminal */
1056 *****/
1057 enter_input:
1058
1059 /* freeup memory taken by sda_i, sda_o, */
1060 /* and cname_i, cname_o names buffers */
1061 cleanup ();
1062
1063 if ( (status = read_query()) < 0 )
1064     goto exit;
1065
1066 /*****
1067 /* BEGIN TRANSACTION */
1068 *****/
1069 exec sql begin work ;
1070
1071 /*****
1072 /* PREPARE the SQL statement */
1073 *****/
1074 exec sql PREPARE S1 from :host1;
1075
1076 if (sqlcode != 0)
1077 { /* display errors/warnings */
1078     printf ("\n"); fflush( stdout );
1079     SQLCADISPLAY ( (int *) &sqlca );
1080     if (sqlcode < 0) /* errors present */
1081     {
1082         exec sql rollback work; /* abort transaction */
1083         goto enter_input; /* try again */
1084     }
1085 }
1086
1087 /*****
1088 /* Allocate input and output sqllda and names buffers */
1089 *****/
1090 out_numvars = sqlsa.u.prepare.output_num;
1091 out_nameslen = sqlsa.u.prepare.output_names_len;
1092 in_numvars = sqlsa.u.prepare.input_num;
1093 in_nameslen = sqlsa.u.prepare.input_names_len;
1094

```

**Example 10-8. Detailed Dynamic SQL Program (page 19 of 22)**

```

1095  if (in_numvars > 0)
1096      if ( (sda_i = allocate_sqlda( in_numvars )) == NULL )
1097          {
1098              printf ("\n**** Error: Memory allocation failure for input
sqlda.\n");
1099              printf ( "                Process stopped.");
1100              fflush (stdout);
1101              goto exit;
1102          }
1103
1104  if (out_numvars > 0)
1105      if ( (sda_o = allocate_sqlda( out_numvars )) == NULL )
1106          {
1107              printf ("\n**** Error: Memory allocation failure for output
sqlda.\n");
1108              printf ( "                Process stopped.");
1109              fflush (stdout);
1110              if (sda_i != NULL) free( (char *) sda_i);
1111              goto exit;
1112          }
1113
1114  if (in_nameslen > 0)
1115      if ( (cname_i = (arrayptr) malloc( in_nameslen )) == NULL )
1116          {
1117              printf("\n**** Error: Memory allocation failure for input names
buffer.");
1118              printf("\n                Process stopped.");
1119              fflush (stdout);
1120              if (sda_i != NULL) free( (char *) sda_i);
1121              if (sda_o != NULL) free( (char *) sda_o);
1122              goto exit;
1123          }
1124
1125  if (out_nameslen > 0)
1126      if ( (cname_o = (arrayptr) malloc( out_nameslen )) == NULL )
1127          {
1128              printf ("\n");
1129              printf ("**** Error: Memory allocation failure for output names
buffer.");
1130              printf ("\n");
1131              printf ( "                Process stopped.");
1132              fflush (stdout);
1133              if (sda_i != NULL) free( (char *) sda_i);
1134              if (sda_o != NULL) free( (char *) sda_o);
1135              if (cname_i != NULL) free( (char *) cname_i );
1136              goto exit;
1137          }
1138
1139  /*****
1140  /* Get information on input variables (if any)          */
1141  *****/
1142  if (in_numvars > 0) {
1143
1144      exec sql DESCRIBE INPUT S1 INTO :sda_i
1145              NAMES INTO :cname_i ;
1146
1147      if (sqlcode != 0)
1148          { /* display error/warnings */
1149              printf ("\n"); fflush( stdout );
1150              SQLCADISPLAY ( (int *) &sqlca );
1151              if (sqlcode < 0) /* errors present */
1152                  {
1153                      exec sql rollback work; /* abort transaction */
1154                      goto enter_input; /* try again */

```

---

**Example 10-8. Detailed Dynamic SQL Program** (page 20 of 22)

---

```

1155     }
1156   }
1157
1158   /*****
1159   /* Input parameter values from terminal */
1160   /* Initialize SQLDA var-ptr to point to input data buffer */
1161   /*****
1162   if ( setupvarbuffers( sda_i ) != 0 )
1163   { printf( "**** Error: Problem in allocating input param buffers.\n");
1164     fflush( stdout);
1165     exec sql rollback work;
1166     goto enter_input;
1167   }
1168
1169   if ( request_invars( sda_i, (char *) cname_i ) < 0 )
1170   {
1171     exec sql rollback work;
1172     goto enter_input;          /* try again */
1173   }
1174
1175   } /* if in_numvars > 0 */
1176
1177   /*****
1178   /* Get information on output variables */
1179   /*****
1180   if ( out_numvars > 0 ) {
1181     exec sql DESCRIBE S1 INTO :sda_o
1182           NAMES INTO :cname_o ;
1183
1184     if (sqlcode != 0)
1185     { /* display error/warnings */
1186       printf("\n"); fflush( stdout );
1187       SQLCADISPLAY ( (int *) &sqlca );
1188       if (sqlcode < 0)          /* errors present */
1189       {
1190         exec sql rollback work; /* abort transaction */
1191         goto enter_input;      /* try again */
1192       }
1193     }
1194   }
1195
1196   /*****
1197   /* Allocate output data buffers and update output sqlda */
1198   /* Initialize SQLDA var-ptr to point to output data buffer */
1199   /*****
1200   if ( setupvarbuffers( sda_o ) != 0 )
1201   { printf( "**** Error: Problem in allocating output buffers\n");
1202     fflush( stdout);
1203     exec sql rollback work;
1204     goto enter_input;
1205   }
1206
1207   } /* if out_numvars > 0 */
1208
1209   if (out_numvars > 0)
1210   { /*****
1211     /* SELECT statement */
1212     /*****
1213

```

---

**Example 10-8. Detailed Dynamic SQL Program (page 21 of 22)**


---

```

1214  /* ----- */
1215  /* Define a cursor name for the statement S1, to be */
1216  /* used later in OPEN, FETCH and CLOSE statements */
1217  /* ----- */
1218  exec sql DECLARE C1 CURSOR for S1 ;
1219
1220  /*-----*/
1221  /* Open the cursor. By this point, all input */
1222  /* parameters must have valid values */
1223  /*-----*/
1224  if (in_numvars > 0)
1225      exec sql OPEN C1 USING DESCRIPTOR :sda_i ;
1226  else
1227      exec sql OPEN C1;
1228
1229  if (sqlcode != 0)
1230      { /* display error/warnings */
1231          printf ("\n"); fflush( stdout );
1232          SQLCADISPLAY ( (int *) &sqlca );
1233          if (sqlcode < 0) /* errors present */
1234              {
1235                  exec sql rollback work; /* abort transaction */
1236                  goto enter_input; /* try again */
1237              }
1238      }
1239
1240  /*-----*/
1241  /* FETCH loop */
1242  /*-----*/
1243  sqlcode = 0;
1244  num_fetches = 0;
1245
1246  while (sqlcode >= 0) {
1247      exec sql fetch C1 USING DESCRIPTOR :sda_o ;
1248
1249      if (sqlcode == 100) /* eof */
1250          { printf( "\n-- %lu row(s) selected.\n", num_fetches);
1251            fflush( stdout);
1252            exec sql close C1 ; /* close cursor */
1253            exec sql commit work;
1254            goto enter_input;
1255          }
1256
1257      /* ----- */
1258      /* Successful FETCH. Display results */
1259      /* ----- */
1260      if (sqlcode >= 0)
1261          {
1262              display_result( sda_o, (char *) cname_o );
1263              num_fetches++; /* increment counter */
1264          }
1265      } /* while loop */
1266
1267      /* ----- */
1268      /* FETCH error. Close cursor. Get next request */
1269      /* ----- */
1270      if (sqlcode < 0)
1271      {

```

---

---

**Example 10-8. Detailed Dynamic SQL Program (page 22 of 22)**

---

```

1272     printf ("\n"); fflush( stdout );
1273     SQLCADISPLAY ((int *) &sqlca);      /* display errors */
1274     exec sql close C1;                  /* close cursor */
1275     exec sql rollback work;
1276     goto enter_input;
1277 }
1278 } /* end: select stmt case */
1279 else
1280 { /*******/
1281     /* Not a SELECT statement. Perform EXECUTE with */
1282     /* USING DESCRIPTOR if there are input variables; */
1283     /* otherwise, perform EXECUTE */
1284     /*******/
1285     if (in_numvars > 0 )
1286     {
1287         exec sql execute S1 using descriptor :*sda_i ;
1288     }
1289     else
1290     {
1291         exec sql execute S1 ;
1292     }
1293
1294     if (sqlcode != 0)
1295     { /* display error/warnings */
1296         printf ("\n"); fflush( stdout );
1297         SQLCADISPLAY ( (int *) &sqlca );
1298         if (sqlcode < 0) /* errors present */
1299         {
1300             exec sql rollback work; /* abort transaction */
1301             goto enter_input; /* try again */
1302         }
1303     }
1304
1305     printf( "\n--- SQL Operation Complete.\n");
1306     fflush( stdout );
1307
1308 } /* end: not a select stmt case */
1309
1310 /* ----- */
1311 /* Successful execution of present query. Commit work. */
1312 /* Process next query */
1313 /* ----- */
1314 exec sql commit work;
1315
1316 goto enter_input;
1317
1318 exit:
1319 printf("\nEnd of current session\n");
1320 fflush (stdout);
1321 } /* end of main */

```

---





## Character Processing Rules (CPRL) Procedures

A C program can call character processing rules (CPRL) procedures to process these collation objects:

- SQL collation—A NonStop 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 compiler

[Table 11-1](#) summarizes the CPRL system procedures. These procedures are listed alphabetically.

---

**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)

|                         |                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------|
| 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 might 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 |

## cextdecs Header File

The `cextdecs` header file contains source declarations for CPRL procedures, which are written in TAL. Use the `#include` directive as shown in this example to copy the declarations from the `cextdecs` header file for the procedures you want to call in your program:

```
#include <cextdecs ( FILE_OPEN_ ,      /
                   WRITEREAD_ ,      /
                   FILE_CLOSE_ ,      /
                   CPRL_INFO_ ,      /
                   CPRL_UPSHIFT_ )> nolist
...
```

## CPRL Return Codes

Each CPRL procedure returns specific codes, which are listed in each procedure description. A return code of zero (0) indicates that the operation was successful. All other CPRL return codes are negative, 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.

```
#include <cextdecs(CPRL_ARE_)>

short CPRL_ARE_ (
    char *classname           /* i */
    ,short classnamelength    /* i */
    ,char *inputstring        /* i */
    ,short inputstringlength   /* i */
    ,long *exceptcharaddr     /* o */
    ,long cprladdr ) ;       /* i */
```

The CPRL\_ARE\_ procedure returns these values:

| 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*

is an array containing the name of the specified character class.

*classnamelength*

is the number of bytes in the character class name *classname*.

*inputstring*

is a string containing the data to be scanned.

*inputstringlength*

is the number of bytes to be scanned in *inputstring*.

*exceptcharaddr*

is set as follows:

- If the call is successful, all scanned characters are in the character class defined by the specified SQL collation or collation object, and *exceptcharaddr* is set as follows:  

$$\text{exceptcharaddr} = \text{inputstring} + \text{inputstringlength}$$
- If -6 is returned, the first character in *inputstring* 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*

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.

```
#include <cextdecs(CPRL_AREALPHAS_)>

short CPRL_AREALPHAS_ (
    char *inputstring          /* i */
    ,short inputstringlength  /* i */
    ,long *exceptcharaddr     /* o */
    ,long cprladdr );        /* i */
```

The CPRL\_AREALPHAS\_ procedure returns these values:

| 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*

is an array containing the string to be scanned.

*inputstringlength*

is the number of bytes to be scanned in *inputstring*.

*exceptcharaddr*

is set as follows:

- If the call is successful, all the scanned characters are in the ALPHAS character class, and *exceptcharaddr* is set as follows:

*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.

*cprraddr*

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.

```
#include <cextdecs(CPRL_ARENUMERICS_)>

short CPRL_ARENUMERICS_ (
    char *inputstring          /* i */
    ,short inputstringlength  /* i */
    ,long *exceptcharaddr     /* o */
    ,long cprraddr );        /* i */
```

The CPRL\_ARENUMERICS\_ procedure returns these values:

| 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*

is an array containing the data to be scanned.

*inputstringlength*

is the number of bytes in *inputstring* to be scanned.

*exceptcharaddr*

is set as follows:

- If the call is successful, all the scanned characters are numeric characters, and *exceptcharaddr* is set as follows:

*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*

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.

```
#include <cextdecs(CPRL_COMPARE1ENCODED_)>

short CPRL_COMPARE1ENCODED_ (
    char *string1          /* i */
    ,short string1length   /* i */
    ,char *string2         /* i */
    ,short string2length   /* i */
    ,short *result         /* o */
    ,long cprladdr );      /* i */
```

The CPRL\_COMPARE1ENCODED\_ procedure returns these values:

| 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*

is an array containing the first string to be compared. *string1* is assumed to be in encoded form.

*string1length*

is the number of bytes in *string1* to be compared.

*string2*

is an array containing the second string to be compared. *string2* is assumed to be in original (not encoded) form.

*string2length*

is the length of *string2*.

*result*

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*

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.

CPRL\_COMPARE\_ is more efficient for isolated compares, because only the necessary part of each string is encoded to do the compare. If the same data is repeatedly compared, use the CPRL\_ENCODE\_ and CPRL\_COMPARE1ENCODED\_ procedures (or CPRL\_ENCODE\_ with binary compares.)

```
#include <cextdecs(CPRL_COMPARE_)>

short CPRL_COMPARE_ (
    char *string1          /* i */
    ,short string1length  /* i */
    ,char *string2        /* i */
    ,short string2length  /* i */
    ,short *result        /* o */
    ,long cprladdr );    /* i */
```

The CPRL\_COMPARE\_ procedure returns these values:

| 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*

is an array containing the first string to be compared.

*string1length*

is the length in bytes of *string1*.

*string2*

is an array containing the second string to be compared.

*string2length*

is the length in bytes of *string2*.

*result*

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*

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.

```
#include <cextdecs(CPRL_COMPAREOBJECTS_)>

short CPRL_COMPAREOBJECTS_ (
    long cprladdr1      /* i */
    ,long cprladdr2 );  /* i */
```



The CPRL\_COMPAREOBJECTS\_ procedure returns these values:

| 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*

is the address of the first SQL collation or collation object.

*cprladdr2*

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 equals 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 when decoded, is aBcDe.

```
#include <cextdecs(CPRL_DECODE_)>

short CPRL_DECODE_ (
    char *encodedstring           /* i */
    ,short encodedstringlength    /* i */
    ,char *decodedstring          /* o */
    ,short decodedstringmaxlength /* i */
    ,short *decodedstringlength   /* o */
    ,long cprladdr );            /* i */
```

The CPRL\_DECODE\_ procedure returns these values:

| 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*

is an array containing the data to be decoded.

*encodedstringlength*

is the number of bytes in *encodedstring* to be decoded.

*decodedstring*

is an array in which CPRL\_DECODE\_ returns the decoded string. Overlapping *encodedstring* and *decodedstring* causes unpredictable results.

*decodedstringmaxlength*

specifies the maximum length of *decodedstring*.

*decodedstringlength*

is the number of bytes of *encodedstring* that were decoded. CPRL\_DECODE\_ pads the remainder of *decodedstring* with blanks up to *decodedstringmaxlength*.

*cprraddr*

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.

```
#include <cextdecs(CPRL_DOWNSHIFT_)>

short CPRL_DOWNSHIFT_ (
    char *inputstring           /* i */
  ,short inputstringlength     /* i */
  ,char *shiftedstring         /* o */
  ,short shiftedstringmaxlength /* i */
  ,short *shiftedstringlength  /* o */
  ,long cprraddr );           /* i */
```

The CPRL\_DOWNSHIFT\_ procedure returns these values:

### 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*

is an array in which CPRL\_UPSHIFT\_ returns the downshifted string.

*inputstringlength*

is the number of bytes to be downshifted in *inputstring*.

*shiftedstring*

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*

specifies the maximum length of *shiftedstring*; it must be greater than equal to *inputstring*.

*shiftedstringlength*

specifies the length of the downshifted string returned in *shiftedstring*.

*cprladdr*

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 produces 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).

```
#include <cextdecs(CPRL_ENCODE_)>

short CPRL_ENCODE_ (
    char *decodedstring           /* i */
    ,short decodedstringlength    /* i */
    ,char *encodedstring          /* o */
    ,short encodedstringmaxlength /* i */
    ,short *encodedstringlength   /* o */
    ,long cprladdr );            /* i */
```

The CPRL\_ENCODE\_ procedure returns these values:

| 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*

is an array containing data to be encoded.

*decodedstringlength*

is the number of bytes in *decodedstring* to be encoded.

*encodedstring*

is an array in which CPRL\_ENCODE\_ returns the encoded string. Overlapping *decodedstring* and *encodedstring* causes unpredictable results.

*encodedstringmaxlength*

specifies the maximum length of *encodedstring*.

*encodedstringlength*

is the number of bytes that were encoded. CPRL\_ENCODE\_ pads the remainder of *decodedstring* with encoded blanks up to *decodedstringmaxlength*.

*cprladdr*

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.

```
#include <cextdecs(CPRL_GETALPHATABLE_)>

short CPRL_GETALPHATABLE_ (
    char *array      /* o */
    ,long cprladdr ); /* i */
```

The CPRL\_GETALPHATABLE\_ procedure returns these values:

| 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*

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*

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.

```
#include <cextdecs(CPRL_GETCHARCLASSTABLE_)>

short CPRL_GETCHARCLASSTABLE_(
    char *array           /* o */
    , long cprladdr       /* i */
    , char *classname     /* i */
    , short classnamelength ); /* i */
```

The CPRL\_GETCHARCLASSTABLE\_ procedure returns these values:

| 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*

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*

is a pointer to the SQL collation or collation object.

*classname*

is the name of the user-specified character class.

*classnamelength*

is the length of *classname* in bytes.

## CPRL\_GETDOWNSHIFTTABLE\_

The CPRL\_GETDOWNSHIFTTABLE\_ procedure extracts downshift information from an SQL collation or collation object.

```
#include <cextdecs(CPRL_GETDOWNSHIFTTABLE_)>

short CPRL_GETDOWNSHIFTTABLE_ (
    char *array          /* o */
    ,long cprladdr );    /* i */
```

The CPRL\_GETDOWNSHIFTTABLE\_ procedure returns these values:

### 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*

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*

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 nonbinary collating sequences.

```
#include <cextdecs(CPRL_GETFIRST_)>

short CPRL_GETFIRST_ (
    char *firststring          /* o */
    ,short firststringmaxlength /* i */
    ,short *firststringlength  /* o */
    ,long cprladdr );         /* i */
```

The CPRL\_GETFIRST\_ procedure returns these values:

### 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*

is an array in which CPRL\_GETFIRST\_ returns the first string.

*firststringmaxlength*

is the maximum length of *firststring*.

*firststringlength*

specifies the number of bytes of *firststring* that were scanned. (If CPRL\_GETFIRST\_ is successful, *firststringmaxlength* and *firststringlength* are equal.)

*cprladdr*

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 nonbinary collating sequences.

```
#include <cextdecs(CPRL_GETLAST_)>

short CPRL_GETLAST_ (
    char *laststring          /* o */
    ,short laststringmaxlength /* i */
    ,short *laststringlength  /* o */
    ,long cprladdr );        /* i */
```

The CPRL\_GETLAST\_ procedure returns these values:

## 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*

is an array in which CPRL\_GETFIRST\_ returns the last string.

*laststringmaxlength*

specifies the maximum length of *laststring*.

*laststringlength*

specifies the number of bytes of *laststring* that were scanned. (If CPRL\_GETLAST\_ is successful, *laststringlength* and *laststringmaxlength* are equal.)

*cprladdr*

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 nonbinary collating sequences.

```
#include <cextdecs(CPRL_GETNEXTINSEQUENCE_)>

short CPRL_GETNEXTINSEQUENCE_ (
    char *inputstring          /* i */
    ,short inputstringlength   /* i */
    ,char *nextstring          /* o */
    ,short nextstringmaxlength /* i */
    ,short *nextstringlength   /* o */
    ,long cprladdr );         /* i */
```

The CPRL\_GETNEXTINSEQUENCE\_ procedure returns these values:

## 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).

*inputstring*

is an array containing the input string.

*inputstringlength*

is the number of bytes in the input string *inputstring*.

*nextstring*

is an array in which CPRL\_GETNEXTINSEQUENCE\_ returns the next string. Overlapping *inputstring* and *nextstring* causes unpredictable results.

*nextstringmaxlength*

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*

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*

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.

```
#include <cextdecs(CPRL_GETNUMTABLE_)>

short CPRL_GETNUMTABLE_ (
    char *array          /* o */
    ,long cprladdr );    /* i */
```

The CPRL\_GETNUMTABLE\_ procedure returns these values:

### 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*

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*

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.)

```
#include <cextdecs(CPRL_GETSPECIALTABLE_) >

short CPRL_GETSPECIALTABLE_ (
    char *array          /* o */
    ,long cprladdr );    /* i */
```

The CPRL\_GETSPECIALTABLE\_ procedure returns these values:

## 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*

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 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*

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.

```
#include <cextdecs(CPRL_GETUPSHIFTTABLE_)>

short CPRL_GETUPSHIFTTABLE_ (
    char *array          /* o */
    ,long cprladdr );    /* i */
```

The CPRL\_GETUPSHIFTTABLE\_ procedure returns these values:

| 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*

is a 256-byte array specified by the user.

If the call is successful, CPRL\_GETALPHATABLE\_ 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*

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.)

```
#include <cextdecs(CPRL_INFO_)>

short CPRL_INFO_ (
    long cprladdr          /* i */
    , [ short *cprlsize    ] /* o */
    , [ short *is1to1      ] /* o */
    , [ short *lengtheningfactor ] /* o */
    , [ short *character set ] /* o */
    , [ short *version     ] ) ; /* o */
```

The CPRL\_INFO\_ procedure returns these values:

**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*

is a pointer to the SQL collation or collation object.

*cprlsize*

is the length in bytes of the SQL collation or collation object.

*islto1*

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*

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.)

*character set*

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*

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.

```
#include <cextdecs(CPRL_READOBJECT_)>

short CPRL_READOBJECT_ (
    short *buffer           /* o */
    ,short bufferlength     /* i */
    ,short *objectlength    /* o */
    ,char *filename         /* i */
    ,short filenamelength   /* i */
    ,long *cprladdr );     /* o */
```

The CPRL\_READOBJECT\_ procedure returns these values:

## 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\_ returns 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*

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*

is the size of *buffer* in bytes.

### *objectlength*

is the actual length in bytes of the collation object read into *buffer*.

*filename*

is the Guardian file name in external format containing the collation object. The file code for *filename* must be 199.

*filenamelength*

is the length in bytes of *filename*.

*cprladdr*

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.

```
#include <cextdecs(CPRL_UPSHIFT_)>

short CPRL_UPSHIFT_ (
    char *inputstring           /* i */
    ,short inputstringlength    /* i */
    ,char *shiftedstring        /* o */
    ,short shiftedstringmaxlength /* i */
    ,short *shiftedstringlength /* o */
    ,long cprladdr );          /* i */
```

The CPRL\_UPSHIFT\_ procedure returns these values:

### 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*

is an array containing the data to be upshifted.

*inputstringlength*

is the number of bytes in *inputstring* to be upshifted.

*shiftedstring*

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*

specifies the maximum length of *shiftedstring*; it must be greater than or equal to *inputstringlength*.

*shiftedstringlength*

specifies the length of the upshifted string returned in *shiftedstring*.

*cprladdr*

is a pointer to the SQL collation or collation object.



# **A** SQL/MP Sample Database

This appendix describes the NonStop SQL/MP sample database included on the product site update tape (SUT). Many examples in this manual (in addition to other SQL/MP manuals) refer to this 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.)

HP distributes the sample database in the ZTSQLMSG subvolume. (Ask your database administrator or system manager for the volume where the ZTSQLMSG subvolume is installed on your system.)

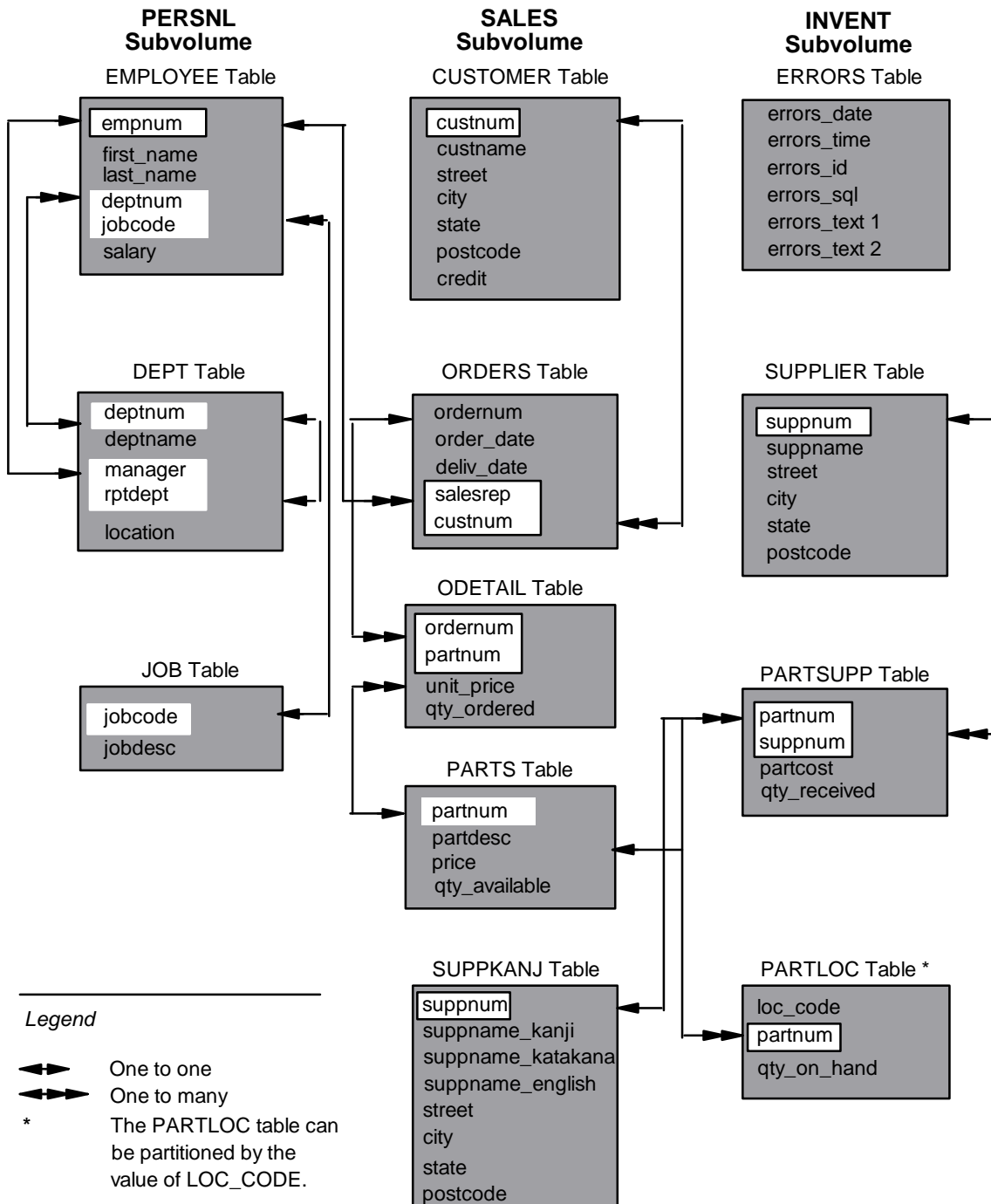
The ZTSQLMSG.DOCUMENT file describes the files in the ZTSQLMSG subvolume. The DOCUMENT file also explains how to create a copy of the sample database. To print the DOCUMENT file, enter this TGAL command at your TACL prompt:

```
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**



VST007.vsd

[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 C TO COPYLIB (DEPT);
```

For more information about SQLCI, see the *SQL/MP Reference Manual*. For a description of the SUPPKANJ table, see the ZTSQMLMSG.DOCUMENT file.

---

**Example A-1. COPYLIB File for Sample Database** (page 1 of 3)

---

```
/* Personnel (PERSNL) */
/* */
#pragma SECTION EMPLOYEE
/* Record Definition for \SYS1.$VOL1.PERSNL.EMPLOYEE */
/* Definition current at 09:53:58 - 11/10/96 */
struct employee_type {
    unsigned short empnum;
    char first_name[16];
    char last_name[21];
    unsigned short deptnum;
    unsigned short jobcode;
    unsigned long salary; /* scale is 2 */
};
#pragma SECTION DEPT
/* Record Definition for \SYS1.$VOL1.PERSNL.DEPT */
/* Definition current at 09:54:00 - 11/10/96 */
struct dept_type {
    unsigned short deptnum;
    char deptname[13];
    unsigned short manager;
    unsigned short rptdept;
    struct {
        short len;
        char val[19];
    } location;
};
#pragma SECTION JOB
/* Record Definition for \SYS1.$VOL1.PERSNL.JOB */
/* Definition current at 09:54:02 - 11/10/96 */
struct job_type {
    unsigned short jobcode;
    struct {
        short len;
        char val[19];
    } jobdesc;
};
/* */
/* Sales (SALES) */
/* */
```

---

---

**Example A-1. COPYLIB File for Sample Database** (page 2 of 3)

---

```

#pragma SECTION CUSTOMER
/* Record Definition for \SYS1.$VOL1.SALES.CUSTOMER          */
/* Definition current at 09:54:03 - 11/10/96                  */
struct customer_type {
    unsigned short  custnum;
    char            custname[19];
    char            street[23];
    char            city[15];
    char            state[13];
    char            postcode[11];
    char            credit[3];
};
#pragma SECTION ORDERS
/* Record Definition for \SYS1.$VOL1.SALES.ORDERS            */
/* Definition current at 09:54:05 - 11/10/96                  */
struct orders_type {
    unsigned long   ordernum;
    long            order_date;
    long            deliv_date;
    unsigned short  salesrep;
    unsigned short  custnum;
};
#pragma SECTION ODETAIL
/* Record Definition for \SYS1.$VOL1.SALES.ODETAIL           */
/* Definition current at 09:54:06 - 11/10/96                  */
struct odetail_type {
    unsigned long   ordernum;
    unsigned short  partnum;
    long            unit_price;                /* scale is 2 */
    unsigned long   qty_ordered;
};
#pragma SECTION PARTS
/* Record Definition for \SYS1.$VOL1.SALES.PARTS             */
/* Definition current at 09:54:08 - 11/10/96                  */
struct parts_type {
    unsigned short  partnum;
    char            partdesc[19];
    long            price;                    /* scale is 2 */
    long            qty_available;
};

```

---

---

**Example A-1. COPYLIB File for Sample Database** (page 3 of 3)

---

```

#pragma SECTION PARTSUPP
/* Record Definition for \SYS1.$VOL1.INVENT.PARTSUPP      */
/* Definition current at 09:54:09 - 11/10/96               */
struct partsupp_type {
    unsigned short partnum;
    unsigned short suppname;
    long partcost; /* scale is 2 */
    unsigned long qty_received;
};
/*
/* Inventory (INVENT)
/*
#pragma SECTION SUPPLIER
/* Record Definition for \SYS1.$VOL1.INVENT.SUPPLIER      */
/* Definition current at 09:54:11 - 11/10/96               */
struct supplier_type {
    unsigned short suppname;
    char suppname[19];
    char street[23];
    char city[15];
    char state[13];
    char postcode[11];
};
#pragma SECTION PARTLOC
/* Record Definition for \SYS1.$VOL1.INVENT.PARTLOC      */
/* Definition current at 09:54:12 - 11/10/96               */
struct partloc_type {
    char loc_code[4];
    unsigned short partnum;
    long qty_on_hand;
};
#pragma SECTION ERRORS
/* Record Definition for \SYS1.$VOL1.INVENT.ERRORS      */
/* Definition current at 09:54:14 - 11/10/96               */
struct errors_type {
    long errors_date
    long errors_time
    long errors_id
    short errors_sql
    char errors_text1[241];
    char errors_text2[241];
};

```

---



# B Memory Considerations

This appendix describes the NonStop SQL internal data structures generated in a C program and the memory considerations for these structures.

Topics include:

- [SQL/MP Internal Structures](#)
- [Using the SQLMEM Pragma](#) on page B-2
- [Estimating Memory Requirements](#) on page B-2
- [Avoiding Memory Stack Overflows](#) on page B-4

## SQL/MP Internal Structures

The C compiler generates internal SQL data structures to maintain information about the SQL statements, directives, and host variables that are used in the program.

[Table B-1](#) lists SQL data structures and when each structure is generated.

---

**Table B-1. SQL/MP Data Structures**

| SQL Data Structure | Statement, Directive, or Host Variable                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLIN              | SQL statement or directive that generates a call to the SQL executor, except the following: <ul style="list-style-type: none"><li>• BEGIN DECLARE SECTION or END DECLARE SECTION directive</li><li>• CONTROL directives</li><li>• INCLUDE directive</li><li>• INVOKE directive</li><li>• WHENEVER directive</li><li>• DECLARE CURSOR statement for static cursors</li><li>• DECLARE CURSOR statement for dynamic cursors that do not use cursor or statement host variables</li></ul> |
| SQLIVARS           | Each input host variable specified in the program                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| SQLOVARS           | Each output host variable specified in the program                                                                                                                                                                                                                                                                                                                                                                                                                                    |

---

## Using the SQLMEM Pragma

For programs that use the large-memory model and are compiled on TNS systems, the SQLMEM pragma specifies where in memory the C compiler should place the SQL internal data structures. Use this syntax for the SQLMEM pragma:

|                       |
|-----------------------|
| SQLMEM { USER   EXT } |
|-----------------------|

### USER

causes the C compiler to allocate the SQL data structures in the user data space, which is the global area addressable with 16 bits. Although the USER option can improve access time to the SQL structures, specify USER only if the global area can hold the specific structures.

### EXT

causes the C compiler to place the SQL data structures in an extended data segment. EXT is the default.

Follow these guidelines when you use the SQLMEM pragma:

- The SQLMEM pragma applies only to the C compiler on TNS systems. The NMC compiler on TNS/R systems ignores this pragma.
- The SQLMEM pragma is valid only for the large-memory model, which is the default for the C compiler. If you specify SQLMEM for the small-memory model (NOXMEM pragma) on a TNS system, the C compiler returns error 172.
- To specify the SQLMEM pragma in your program, you must first specify the SQL pragma. Otherwise, the C compiler returns error 173 (illegal SQLMEM option).
- Use the SQLMEM pragma as many times as necessary in your program to control the placement of SQL data structures.

## 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 in a program's extended data segment.

Some statements require no extra extended memory, but other statements generate a run-time call to the SQL executor and use the extra memory. The SQL executor uses extended memory to run and uses the memory shown in this table for parameters and data structures.



These structures are shared by all SQL statements and directives in a program:

| Structure | Bytes       | Description                                                                                                                                        |
|-----------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| SQLCA     | 430         | Count once if you specify the INCLUDE SQLCA directive.                                                                                             |
| SQLSA     | 838 or 1790 | Count once if you specify the INCLUDE SQLSA directive. A version 330 or later SQLSA structure is 1790 bytes; older SQLSA structures are 838 bytes. |

Use [Table B-2](#) to estimate the memory used by each SQL statement and directive.

**Table B-2. Virtual Memory Requirements for SQL Statements**

| Bytes Required                               | Description                                                                                       |
|----------------------------------------------|---------------------------------------------------------------------------------------------------|
| 72                                           | Base value for a statement with no host variables                                                 |
| + 4 + (24 * number of input host variables)  | Required for a statement with input host variables                                                |
| + 4 + (24 * 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 use [Table B-2](#):

- Count a host variable once per occurrence.
- Count 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

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 in [Table B-2](#) on page B-3.

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 the extended data segment, follow these guidelines:

- Declare only the host variables that your program actually requires.
- Declare all host variables in one Declare Section, if possible. The system allocates the host variables contiguously in one or more pages, rather than allocating each host variable in a separate page.
- Run SQL statements in listing order as often as possible. Thus, 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.

## Avoiding Memory Stack Overflows

To avoid memory stack overflows for most SQL statements, the SQL executor needs at least 3000 words of available stack space. To calculate the approximate stack space that should be available to run an SQL statement, use this formula:

$$\text{Stack Space (words)} = 3000 + 300 * (\text{number of referenced tables} - 3)$$

For example, using this formula, an SQL statement that refers to five tables needs approximately 3600 words of stack space:

$$\begin{aligned} \text{Stack Space (words)} &= 3000 + (300 * (5-3)) \\ &= 3600 \end{aligned}$$

The SQL executor handles a stack overflow caused by an SQL statement as follows:

- Less than 1024 words of stack space are available. (The limit of 1024 words is an arbitrary number that is used to prevent problems for existing applications, and might be increased in a future RVU.)

If there is enough stack space available to call an error handling routine, the SQL executor returns SQL error 8003. If there is not enough stack space to call the routine, the executor abends without returning a message.

- At least 1024 words of stack space are available.

If a stack overflow occurs, the executor traps (trap number 3), sends a message to the EMS collector process (\$0), and then abends. You can read the EMS event log for this message. For a description of the SQL/MP messages sent to the \$0 process, see the *SQL/MP Messages Manual*. If a call to the file system causes a stack overflow, the SQL executor returns SQL error 8003 and file-system error 22.

You can prevent a stack overflow in a C program by following these guidelines:

- Use the large-memory model. Specify the XMEM pragma (the default) to select the large-memory model and the XVAR pragma (also the default) to direct the C compiler to allocate static aggregates in extended memory.
- Allocate SQL internal data structures (SQLIN, SQLIVARS, and SQLOVARS) in extended memory (which is the default for the large-memory model). The SQLMEM pragma (USER or EXT) controls the placement of the SQL structures; SQLMEM EXT (extended memory) is the default.
- Use the Binder SET EXTENDSTACK command to extend the user stack space. For more information, see the *Binder Manual*.
- When you start the program, increase the number of data pages for the program using one of these options:
  - Interactively, specify the MEM option to increase the data pages in the TACL RUN command. For more information, see the *TACL Reference Manual*.
  - Programmatically, set the *memory-pages* parameter of the NEWPROCESS or PROCESS\_CREATE\_ system procedure. For more information, see the *Guardian Procedure Calls Reference Manual*.



# 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
- [Skipping Unavailable Partitions](#) on page C-3

Local autonomy in a network-distributed database ensures that a program can access data on the local node, regardless of the availability of remote SQL objects. 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.
- Use TACL DEFINES.
- Use current statistics.
- Skip unavailable partitions.

For collations, NonStop 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 named 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 accesses 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 data are available, the SQL compilation is successful.

The next example uses the concept of maximizing local autonomy. The parts table is a partitioned table that resides on these 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";
```

The program running on \NEWYORK uses a DEFINE to associate the PARTS table with the first partition located at \NEWYORK.

If \PARIS is unavailable at compile time, the SQL compiler can still compile the program 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 SQL executor invokes the SQL compiler to automatically recompile the statement. The SQL compiler determines an execution plan that does not use the index IXPART but sequentially scans the rows in the first partition to find all parts that have "V8 DISK OPTION" in the PARTDESC column.

## Using TACL DEFINES

By using TACL DEFINES in a program to refer to tables and associating those DEFINES with local partitions, you increase the number of successful compilations of programs that access a distributed database. All SQL compilations are affected, including explicit compilations and automatic recompilations.

## Using Current Statistics

For a partitioned table to have local autonomy, the UPDATE STATISTICS statement must be run 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 eliminates both these problems.

# Skipping Unavailable Partitions

Use the SKIP UNAVAILABLE PARTITION option of the CONTROL TABLE directive to cause NonStop 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. (NonStop 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 C Programs

A C program developed for NonStop SQL/MP version 1 or version 2 software can run on SQL/MP version 300 (or later) software without any changes to its embedded SQL statements or directives. However, to use new SQL features, you must modify and recompile the program.

---

**Note.** A D20 (or later) C compiler requires that a C program comply to the ISO/ANSI C standard. For information about converting a program to follow this standard, see the *C/C++ Programmer's Guide*.

Also, a C-series program can run at a low PIN on a D-series system without any changes. However, for a C-series program to use D-series features (for example, to run at a high PIN), you might need to convert certain parts of the program. For information about converting a C-series program to use D-series features, see the *Guardian Application Conversion Guide*.

---

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 future PVUs of NonStop SQL/MP. Follow these guidelines if you are converting an existing C 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 the SQL structures, even if you require version 1 or version 2 structures. To generate version 300 or later structures, you must use the INCLUDE STRUCTURES directive. For more information, see [Section 9, Error and Status Reporting](#).
- If you allocate SQL data structures at run time, use the compiler-generated length identifiers (for example, SQLSA\_LEN for the length of an SQLSA structure) to specify the memory to allocate. (In some cases, you can also use a C function to generate the length of a structure.) Using the compiler-generated length identifiers can reduce the impact on a program if the size of an SQL data structure changes in a future PVU.
- Use the system-generated eye-catcher identifiers to initialize the eye-catcher fields. Do not hard code eye-catcher values or write code that depends on hard-coded values. The eye-catcher values can change in a new PVU.
- Use the SQLCAGETINFOLIST procedure to return information from the SQLCA structure. Do not access this structure directly. HP reserves the right to change the SQLCA structure in future PVUs.

[Table D-1](#) lists the changes to the SQL data structures and the changes that occurred with each version of NonStop SQL/MP.

---

**Table D-1. Changes to SQL Data Structures**

| Version                | Size,<br>Bytes | Eye-Catcher<br>Value | New Fields                                                                                                  |
|------------------------|----------------|----------------------|-------------------------------------------------------------------------------------------------------------|
| <b>SQLCA Structure</b> |                |                      |                                                                                                             |
| 1, 2, Š 300            | 430            | CA                   | None                                                                                                        |
| <b>SQLSA Structure</b> |                |                      |                                                                                                             |
| 1                      | 838            | SA                   | –                                                                                                           |
| 2                      | 838            | SA                   | None                                                                                                        |
| 300 - 325              | 838            | SA                   | output_collations_len                                                                                       |
| Š 330                  | 1790           | SA                   | master_executor_elapsed_time<br>total_esp_cpu_time<br>total_sortprog_cpu_time<br>vsbb_write<br>vsbb_flushed |
| <b>SQLDA Structure</b> |                |                      |                                                                                                             |
| 1                      | Variabl<br>e   | DA                   | –                                                                                                           |
| 2                      | Variabl<br>e   | D1                   | precision, null_info, ind_ptr                                                                               |
| Š 300                  | Variabl<br>e   | D1                   | cprl_ptr, user-defined collation buffer                                                                     |

---

## Generating SQLDA Structures

If an existing C program generates SQLDA structures and you are converting the program 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 2 SQLDA structure
- A version 300 (or later) SQLDA structure and a version 1 or 2 SQLDA structure

## Generating a Version 300 (or Later) 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 C 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 and specify the version you want. For example, this directive generates a version 315 structure:

```
EXEC SQL INCLUDE STRUCTURES ALL VERSION 315;
```

Or specify only the SQLDA VERSION 315 option:

```
EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 315;
```

5. Add the necessary executable statements to process the version 310 SQLDA structure. For the layout of a version 300 (or later) 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 C 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. (A 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;
```

Or specify only the SQLDA VERSION 2 option:

```
EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 2;
```

[Example D-1](#) shows a version 2 SQLDA structure.

---

#### Example D-1. Version 2 SQLDA Structure

```
#define SQLDA_EYE_CATCHER "D1"  /* can have _R2 appended */
struct SQLDA_TYPE                /* can have _R2 appended */
{
    char   eye_catcher[2];
    short  num_entries;
    struct SQLVAR_TYPE            /* can have _R2 appended */
    {
        short  data_type;
        short  data_len;
        short  precision;
        short  null_info;
        long   var_ptr;
        long   ind_ptr;
        long long reserved;
    } sqlvar[sqlvar-count];
} sqlda-name;
char names-buffer-name[ length + 1 ];
```

---

[Table D-2](#) describes the fields in a version 2 SQLDA structure.

---

**Table D-2. Version 2 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 | 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. |
| data_type   | Data type of the parameter or output variables. For the table of data type values, see <a href="#">Section 10, Dynamic SQL Operations</a> .                                          |

---

**Table D-2. Version 2 SQLDA Structure Fields** (page 2 of 2)

| Field Name | Description                                                                                                                                                                                       |                                                                                                                                                                                               |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| data_len   | data_len depends on the data type:                                                                                                                                                                |                                                                                                                                                                                               |
|            | Fixed-length character                                                                                                                                                                            | Number of bytes in the string.                                                                                                                                                                |
|            | Variable-length character                                                                                                                                                                         | Maximum number of bytes in the string.                                                                                                                                                        |
|            | Decimal numeric                                                                                                                                                                                   | Bits 0:7 contain the decimal scale.<br>Bits 8:15 contain the byte length of the item.                                                                                                         |
|            | Binary numeric                                                                                                                                                                                    | Bits 0:7 contain the decimal scale.<br>Bits 8:15 contain the byte length of the item (2, 4, or 8).                                                                                            |
| precision  | Date-time or INTERVAL                                                                                                                                                                             | Bits 0:7 contain a value for the range date-time fields. For the table of values, see <a href="#">Section 10, Dynamic SQL Operations</a> .<br>Bits 8:15 contain the storage size of the item. |
|            | Binary numeric                                                                                                                                                                                    | Numeric precision.                                                                                                                                                                            |
| null_info  | For input parameters                                                                                                                                                                              | Bits 0:7 contain the leading field precision.<br>Bits 8:15 contain the fraction precision (or 0, if the fraction field is not included).                                                      |
|            | For output columns                                                                                                                                                                                | A negative integer if the column permits null values.                                                                                                                                         |
| var_ptr    | For input parameters                                                                                                                                                                              | A negative integer if the row returned is null.                                                                                                                                               |
|            | Extended address of the actual data (value of input parameter or column). NonStop SQL/MP does not return var_ptr; a program must initialize var_ptr to point to the input and output data buffers |                                                                                                                                                                                               |
| ind_ptr    | Address of a flag that indicates whether a parameter or column is actually null.                                                                                                                  |                                                                                                                                                                                               |
|            | For input parameters                                                                                                                                                                              | A negative integer if the column permits null values.                                                                                                                                         |
|            | For output columns                                                                                                                                                                                | A negative integer if the row returned is null.                                                                                                                                               |
|            | NonStop SQL/MP sets the ind_ptr location to -1 if the column value is null.                                                                                                                       |                                                                                                                                                                                               |
|            | If a program does not process null values, set the ind_ptr location 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 C 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;
```

Or specify only the SQLDA VERSION 1 option:

```
EXEC SQL INCLUDE STRUCTURES SQLDA VERSION 1;
```

[Example D-2](#) shows a version 1 SQLDA structure.

---

## Example D-2. Version 1 SQLDA Structure

```
#define SQLDA_EYE_CATCHER "DA" /* can have _R1 appended */
struct SQLDA_TYPE              /* can have _R1 appended */
{
    char   eye_catcher[2];
    short  num_entries;
    struct SQLVAR_TYPE          /* can have _R1 appended */
    {
        short data_type;
        short data_len;
        short null_info;
        long  var_ptr;
        long  reserved;
    } sqlvar[sqlvar_count];
} sqlda-name;
char names-buffer-name[ length + 1 ];
```

---

[Table D-3](#) describes the fields in a version 1 SQLDA structure.

---

**Table D-3. Version 1 SQLDA Structure Fields** (page 1 of 2)

| Field Name  | Description                                                                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| eye_catcher | 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 | 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                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>data_type</code> | Data type of the parameter or output variable. For the table of data type values, see <a href="#">Section 10, Dynamic SQL Operations</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <code>data_len</code>  | <p><code>data_len</code> depends on the data type:</p> <p>Fixed-length character    Number of bytes in the string.</p> <p>Variable-length character   Maximum number of bytes in the string.</p> <p>Decimal numeric            Bits 0:7 contain the decimal scale.<br/>Bits 8:15 contain the byte length of the item.</p> <p>Binary numeric             Bits 0:7 contain the decimal scale.<br/>Bits 8:15 contain the byte length of the item (2, 4, or 8).</p> <p>Date-time or INTERVAL    Bits 0:7 contain a value specifying the range of date-time fields. For the table of values, see <a href="#">Section 10, Dynamic SQL Operations</a>.<br/>Bits 8:15 contain the storage size of the item.</p> |
| <code>var_ptr</code>   | Extended address of the actual data (value of input parameter or column). NonStop SQL/MP does not return <code>var_ptr</code> ; a program must initialize <code>var_ptr</code> to point to the input and output data buffers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## Using a Combination of SQLDA Structures

Version 300 (or later) SQL/MP software does not support different versions of SQLDA structures in the same compilation unit. If a program requires more than one SQLDA structure in a compilation unit, convert all SQLDA structures to version 315. However, to use a combination of SQLDA structures (for example, a version 2 structure and a version 315 structure), follow these steps:

1. Separate the program into different compilation units so that the version 315 SQLDA structure and the supporting executable statements are in a different compilation unit than the version 2 (or version 1) SQLDA structure and its executable 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 C program developed for NonStop SQL/MP version 1 or version 2 software to use version 300 (or later) features and to run on NonStop SQL/MP version 300 (or later) software, consider making these changes in the program for compatibility with future NonStop 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 might generate a run-time error.

If you call any of these procedures, consider modifying the program as follows:

| 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 pragma and the INCLUDE SQLDA directive might not be supported in future PVUs.

Consider modifying the 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 the program to use version 300 (or later) SQLDA structures. Remove the RELEASE1 or RELEASE2 option from the SQL pragma or the INCLUDE SQLDA directive.

For more information about the INCLUDE STRUCTURES directive, see [Section 9, Error and Status Reporting](#).



# Index

## A

### Accelerator

- effect on SQL validity [8-2](#)
- running on object file [1-5](#), [8-2](#)
- running on program file [6-13](#), [6-14](#)

### Access authority

- DELETE statement [4-23](#)
- FETCH statement [4-20](#)
- OPEN statement [4-19](#)
- SELECT statement [4-21](#)
- SQL compilation requirements [6-13](#)
- SQL cursor [4-16](#), [4-18](#)
- UPDATE statement [4-22](#)

### Access path

- EXPLAIN utility [6-16](#), [6-27](#)
- local autonomy [8-6](#)
- RECOMPILE option [6-17](#)
- SQL compiler function [6-13](#)
- unavailable [8-8](#)
- valid programs [8-1](#)

### ADD command, Binder program [6-12](#)

### ADD CONSTRAINT statement, program invalidation [8-5](#)

### add\_define OSS utility [6-29](#)

### Aggregate functions [9-9](#)

### ALLOCATE attribute, similarity check rules [8-12](#)

### ALTER INDEX statement

- error 8204 [4-3](#)
- program invalidation [8-5](#)

### ALTER TABLE statement

- error 8204 [4-3](#)
- program invalidation [8-4](#)
- similarity check considerations [8-13](#)

### ALTER VIEW statement [4-3](#)

### Altering SQL file attribute [8-3](#)

### Arguments, C compiler, RUN command [7-3](#)

### ASSIGN command, TACL [7-2](#)

### Asterisk (\*)

- with pointer as host variable [2-6](#)
- with similarity check [8-13](#)

### Attributes, SQL file [8-3](#)

### AUDIT attribute

- altering and automatic recompilation [8-3](#)
- similarity check rules [8-12](#)

### AUDITCOMPRESS attribute, similarity check rules [8-12](#)

### Authority requirements for program execution [7-1](#)

### Automatic SQL recompilation

- causes [8-6](#)
- collation [8-3](#)
- functions [8-6](#)
- performance considerations [B-4](#)

## B

### Backslash (\), OSS shell escape character [6-29](#)

### Base table

- See Table, SQL

### BEGIN DECLARE SECTION directive [1-2](#), [2-1](#)

### BEGIN WORK statement [10-27](#)

### BIND command [6-12](#)

### Binder program

- ADD command [6-12](#)
- BIND command [6-12](#)
- binding object files [6-11](#)
- BUILD command [6-12](#)
- C programs [6-11](#)
- CHANGE command [7-5](#)
- description [1-5](#)
- effect on SQL validity [8-2](#)
- SELECT command [6-12](#)
- SET EXTENDSTACK command [B-5](#)

Binder program (continued)  
     SQL compiler [6-22](#)  
     STRIP command [6-12](#)  
 BINSERV option, PARAM command [6-22](#)  
 BROWSE ACCESS with SELECT statement [4-4](#)  
 BUFFERED attribute, similarity check rules [8-12](#)  
 BUILD command, Binder program [6-12](#)

## C

C comments, Declare Section [2-2](#)  
 C compiler  
     determining version [6-36](#), [9-3](#)  
     OSS environment [6-30](#)  
     pragmas  
         RUNNABLE [6-9](#), [6-12](#)  
         SQL [3-2](#), [6-7](#), [10-23](#)  
         SQLMEM [3-7](#), [B-2](#), [B-5](#)  
         SYSTYPE [6-9](#)  
         XMEM [10-18](#), [B-5](#)  
         XVAR [B-5](#)  
     RUN command arguments [7-3](#)  
     WHENEVER directive pseudocode [9-6](#)  
 C language  
     compiler [3-2](#)  
     program development [1-1](#)  
 c89 utility  
     Accelerator [6-31](#)  
     Binder program [6-31](#)  
     C compiler [6-30](#)  
     SQL compiler [6-32](#)  
     version considerations [6-33](#)  
 CALL format, WHENEVER directive [9-8](#)  
 CAST function [2-5](#)  
 Catalog  
     authority for program execution [7-1](#)  
     CHECK options [8-10](#)  
     SQLGETCATALOGVERSION [5-18](#)  
     version considerations [8-10](#)

CATALOG clause, SQL compiler [6-6](#), [6-15](#)  
 CATALOG TACL DEFINE [6-6](#), [6-15](#)  
 cextdecs file  
     CPRL procedures [11-2](#)  
     dynamic SQL applications [10-23](#), [10-36](#)  
     header file [5-2](#)  
     JULIANTIMESTAMP procedures [4-10](#)  
     SQLCA structure [9-12](#)  
     SQL/MP procedures [1-4](#)  
     system procedures [5-2](#)  
 CHAR data type, host variable declaration [2-7](#)  
 Character data  
     array as host variable [2-7](#)  
     Corresponding SQL and C data types [2-3](#)  
     INSERT statement [2-8](#)  
     INVOKE directive [2-19](#)  
     SELECT statement [2-7](#)  
     VARCHAR data type [2-9](#)  
 Character processing rules (CPRL) procedures  
     CPRL\_AREALPHAS\_ [11-4](#)  
     CPRL\_ARENUMERICS\_ [11-5](#)  
     CPRL\_ARE\_ [11-3](#)  
     CPRL\_COMPARE1ENCODED\_ [11-6](#)  
     CPRL\_COMPAREOBJECTS\_ [11-8](#)  
     CPRL\_COMPARE\_ [11-7](#)  
     CPRL\_DECODE\_ [11-9](#)  
     CPRL\_DOWNSHIFT\_ [11-10](#)  
     CPRL\_ENCODE\_ [11-11](#)  
     CPRL\_GETALPHATABLE\_ [11-12](#)  
     CPRL\_GETCHARCLASSTABLE\_ [11-13](#)  
     CPRL\_GETDOWNSHIFTTABLE\_ [11-14](#)  
     CPRL\_GETFIRST\_ [11-15](#)  
     CPRL\_GETLAST\_ [11-16](#)  
     CPRL\_GETNEXTINSEQUENCE\_ [11-17](#)

## Character processing rules (CPRL) procedures (continued)

CPRL\_GETNUMTABLE\_ [11-18](#)  
 CPRL\_GETSPECIALTABLE\_ [11-19](#)  
 CPRL\_GETUPSHIFTTABLE\_ [11-20](#)  
 CPRL\_INFO\_ [11-20](#)  
 CPRL\_READOBJECT\_ [11-22](#)  
 CPRL\_UPSHIFT\_ [11-23](#)

## CHAR\_AS\_ARRAY option, SQL pragma [2-7](#), [6-7](#)

## CHAR\_AS\_STRING option, SQL pragma [6-7](#)

## CHECK clause, SQL compiler [6-18](#)

## CHECK option syntax [8-10](#)

## CLEARONPURGE attribute, similarity check rules [8-12](#)

## CLOSE statement [4-24](#)

## CLOSE TABLES option, FREE RESOURCES statement [4-2](#), [4-24](#)

## Closing tables and views [4-2](#)

## COBOL as host language [1-1](#)

## Coding rules for embedding SQL statements [3-1](#)

## Collation

automatic SQL recompilation [8-3](#)

CPRL\_COMPAREOBJECTS\_  
procedure [8-15](#)

similarity check [8-15](#)

## Collation buffer, determining length [10-7](#)

## Collector process, EMS [B-5](#)

## Colon (:) with host variable [1-2](#), [2-6](#)

## Column headings, similarity check rules [8-12](#)

## Comments

Declare Section [2-2](#)

similarity check rules [8-12](#)

## COMMIT WORK statement [10-28](#)

## Communications area, SQL

See SQLCA structure

## Compilation

automatic recompilation [8-5](#)

C compiler syntax [6-9](#)

## Compilation (continued)

C source file [1-5](#)

dynamic SQL statements [6-23](#)

explicit SQL [6-13](#), [6-14](#)

## COMPILE clause, SQL compiler [6-20](#)

## CONTROL TABLE directive [C-3](#)

## Conversational interface

See SQL Conversational Interface  
(SQLCI)

## Conversion, between SQL and C data [2-5](#)

## CONVERTTIMESTAMP function [4-10](#)

## COPY command and lost open error [4-3](#)

## Copying SQL files, effect on SQL validity [8-2](#)

## CPRL\_AREALPHAS\_ procedure [11-4](#)

## CPRL\_ARENUMERICS\_ procedure [11-5](#)

## CPRL\_ARE\_ procedure [11-3](#)

## CPRL\_COMPARE1ENCODED\_ procedure [11-6](#)

## CPRL\_COMPAREOBJECTS\_ procedure [8-15](#), [11-8](#)

## CPRL\_COMPARE\_ procedure [11-7](#)

## CPRL\_DECODE\_ procedure [11-9](#)

## CPRL\_DOWNSHIFT\_ procedure [11-10](#)

## CPRL\_ENCODE\_ procedure [11-11](#)

## CPRL\_GETALPHATABLE\_ procedure [11-12](#)

## CPRL\_GETCHARCLASSTABLE\_ procedure [11-13](#)

## CPRL\_GETDOWNSHIFTTABLE\_ procedure [11-14](#)

## CPRL\_GETFIRST\_ procedure [11-15](#)

## CPRL\_GETLAST\_ procedure [11-16](#)

## CPRL\_GETNEXTINSEQUENCE\_ procedure [11-17](#)

## CPRL\_GETNUMTABLE\_ procedure [11-18](#)

## CPRL\_GETSPECIALTABLE\_ procedure [11-19](#)

## CPRL\_GETUPSHIFTTABLE\_ procedure [11-20](#)

## CPRL\_INFO\_ procedure [11-20](#)

## CPRL\_READOBJECT\_ procedure [11-22](#)

## CPRL\_UPSHIFT\_ procedure [11-23](#)

CREATE CONSTRAINT statement [4-3](#)  
 CREATE INDEX statement  
     NO INVALDATE option [8-3](#)  
     program invalidation [8-5](#)  
 Creation timestamp, similarity check rules [8-12](#)  
 C-series Guardian operating system [7-5](#)  
 CURRENTDEFINES, SQL compiler option [6-15](#)  
 Cursor operations  
     CLOSE statement (dynamic) [10-27](#)  
     CLOSE statement (static) [4-24](#)  
     DECLARE CURSOR statement [4-18](#)  
     DELETE statement [4-23](#)  
     dynamic SQL cursors  
         declaration [10-27](#)  
         description of [10-20](#)  
         opening [10-27](#)  
     FETCH statement [4-15](#), [4-20](#)  
     foreign cursors [4-24](#)  
     guidelines [4-22](#), [10-20](#)  
     host variables [4-19](#)  
     initializing [4-19](#)  
     lost open error [4-4](#)  
     OPEN statement [4-15](#), [4-19](#)  
     process access ID (PAID) requirements  
         DECLARE CURSOR statement [4-18](#)  
         DELETE statement [4-23](#)  
         description [4-22](#)  
         FETCH statement [4-20](#)  
         OPEN statement [4-19](#)  
         SELECT statement [4-21](#)  
         SQL objects [4-16](#)  
         UPDATE statement [4-22](#)  
     SELECT statement [4-21](#)  
     stability of cursor [4-17](#)  
     Virtual sequential block buffering (VSBB) [4-17](#)  
     WHERE clause [4-21](#)

## D

Data conversion between C and SQL data types [2-5](#)  
 Data declarations  
     BEGIN DECLARE SECTION directive [2-1](#)  
     END DECLARE SECTION directive [2-1](#)  
     statements [1-3](#)  
     tables and views [2-19](#)  
 Data Definition Language (DDL)  
     SQL statements [1-3](#)  
     Tandem statements [10-37](#)  
 Data Manipulation Language (DML) statements [1-3](#)  
 Data status language (DSL) statements [1-3](#)  
 Data structures, SQL  
     description [B-1](#)  
     placing in memory [B-2](#)  
 Data types  
     C [2-3](#), [2-4](#)  
     conversion between SQL and C [2-5](#)  
     SQL [2-3](#), [2-4](#)  
 Database, sample [A-1](#)  
 Data, SQL  
     DELETE statement [4-12](#), [4-23](#)  
     FETCH statement [4-20](#)  
     INSERT statement [4-8](#)  
     SELECT statement [4-4](#)  
     type correspondence (SQL and C) [2-3](#), [2-4](#)  
     UPDATE statement [4-10](#)  
 DATEFORMAT clause  
     example [2-14](#)  
     INVOKE directive [2-14](#)  
 Date-time data type with INVOKE directive [2-14](#)  
 DDL operations, invalidating [8-4](#)

## Debugging

FORCE option [6-16](#), [6-23](#)RUND command [7-3](#)Decimal data type as host variable [2-11](#)

## Declarations, SQLDA

SQLDA\_EYE\_CATCHER [10-5](#)SQLDA\_HEADER\_LEN [10-5](#)SQLDA\_NAMESBUF\_OVHD\_LEN [10-5](#)SQLDA\_SQLVAR\_LEN [10-5](#)DECLARE CURSOR statement [4-18](#)Declare Section [1-2](#), [2-1](#)dec\_to\_longlong C routine [2-11](#)

## DEFAULTS DEFINE

See =\_DEFAULTS DEFINE, TACL

DEFINE format, EXPLAIN report [6-27](#)

## DEFINES option

EXPLAIN utility [6-16](#)SQL compiler [6-16](#)

## DEFINES, TACL

automatic recompilation [8-7](#)INVOKE directive [2-19](#)local autonomy [C-2](#)SQL program file [7-2](#)

## DELETE statement

automatic recompilation [8-3](#)multiple rows [4-13](#)with a cursor [4-23](#)del\_define OSS utility [6-29](#)

## Descriptor area, SQL

See SQLDA structure

DETAIL option, FILEINFO command [8-1](#)

## Directives

See SQL/MP directives

## Disk process (DP2)

SQLCADISPLAY [5-3](#)SQLCAFSCODE [5-8](#)SQLCATOBUFFER [5-14](#)Distributed database, maximizing local autonomy [C-1](#)Double hyphen (--) in SQL statements [3-1](#)Double quotes (") in SQL statements [3-1](#)

## DP2

See Disk process (DP2)

## DROP CONSTRAINT statement

error 8204 [4-3](#)program invalidation [8-5](#)

## DROP INDEX statement

error 8204 [4-3](#)program invalidation [8-5](#)

## DROP TABLE statement

error 8204 [4-3](#)program invalidation [8-5](#)

## DROP VIEW statement

error 8204 [4-3](#)program invalidation [8-5](#)D-series Guardian operating system [7-4](#)DUPLICATE command, FUP [8-2](#)Duplicating SQL files, effect on SQL validity [8-2](#)Dynamic memory allocation [10-18](#)

## Dynamic SQL

compilation [8-5](#)conversational interface [10-1](#)description [1-6](#)dynamic SQL statements [1-3](#), [10-2](#)getting information [10-3](#)input parameters [10-11](#)names buffer [10-3](#)null values [10-16](#)output variables [10-12](#)overview [10-1](#)parameter list [10-12](#)Pathway server [10-36](#)SQLDA structure [10-3](#)SQLSA statistics [9-13](#)statements [10-2](#)Dynamic SQL, statement compilation [6-23](#)

## E

### Embedded SQL statements

- advantages [1-1](#)
- description [1-3](#)
- in C source file [1-3](#), [3-1](#)
- overview [1-1](#)

### Empty section in SQL program [6-21](#)

### EMS collector process [B-5](#)

### END DECLARE SECTION directive [1-2](#), [2-1](#)

### Enscribe database

- memory use by program [B-2](#)
- utilities protection for SQL objects [8-1](#)

### Enscribe I/O [5-6](#), [5-21](#)

### Error and status reporting

- description [1-5](#), [9-1](#)
- display format control [5-5](#), [5-15](#)
- SQL procedures [5-1](#)
- SQLCADISPLAY procedure [5-3](#)
- SQLCAFSCODE procedure [5-8](#)
- SQLCATOBUFFER procedure [5-14](#)
- sqlcode [9-4](#)
- SQLMSG file [5-2](#)
- SQLSA structure [9-13](#)
- WHENEVER directive [9-6](#)

### Errors and warnings

- SQL/MP, SQL compiler [6-23](#)

### Errors and warnings, SQL/MP

- DELETE statement [4-13](#), [4-23](#)
- disk-process errors [5-8](#)
- FETCH statement [4-20](#)
- file-system errors [5-8](#)
- INSERT statement [4-8](#)
- operating system errors [5-8](#)
- run-time SQL recompilation [8-9](#)
- UPDATE statement [4-11](#)

### EXEC SQL keywords [3-1](#)

### EXECUTE IMMEDIATE statement

- description [1-6](#)

### EXECUTE IMMEDIATE statement, SQL compilation errors [6-23](#)

### EXECUTE statement [1-6](#)

### Executing a C program [1-5](#), [7-1](#)

### Execution plan

- EXPLAIN report [6-27](#)
- optimized by SQL compiler [6-13](#)
- optimized by statistics [6-23](#)
- SQL compiler function [6-13](#)

### EXPLAIN utility

- EXPLAIN DEFINES report [6-27](#)
- EXPLAIN PLAN report [6-27](#)
- SQL compiler option [6-16](#)
- SQLCOMP DEFINES option [7-2](#)
- Wverbose flag [6-33](#)

### Explicit SQL compilation

- description [1-5](#)

### Explicit SQL compilation, SQLCOMP command [6-14](#)

### EXT option, SQLMEM pragma [B-2](#)

### Extended data segment, specifying default [B-2](#)

### EXTENT attribute, similarity check rules [8-12](#)

### eye\_catcher field in SQLDA, initializing [10-5](#), [10-29](#)

## F

### FastSort program [5-3](#), [5-14](#)

### FETCH statement [4-15](#), [4-22](#), [10-12](#)

### File attributes, SQL

- effect of altering [8-3](#)
- similarity check rules [8-12](#)

### File label, SQL program

- inconsistency with catalog [8-4](#)
- SQL validation [8-1](#)

### File number of SQLMSG file [5-4](#), [5-15](#)

### File Utility Program (FUP)

- DUPLICATE command [8-2](#)
- FILEINFO command [8-1](#)



## FILEINFO command

FUP [8-1](#)SQLCI [8-1](#)

## File-system errors

SQLCADISPLAY procedure [5-3](#)SQLCAFSCODE procedure [5-8](#)SQLCATOBUFFER procedure [5-14](#)FILE\_GETINFOBYNAME\_ procedure [5-2](#)FILE\_GETINFOLISTBYNAME\_  
procedure [5-2](#)FILE\_GETINFOLIST\_ procedure [5-2](#)FILE\_GETINFO\_ procedure [5-2](#)First error flag, SQLCAFSCODE  
procedure [5-8](#)Fixed-length character data, host variable  
declaration [2-7](#)Fixed-point numeric data, host variable  
declaration [2-11](#)Flag, SQL object file [8-1](#)FOR UPDATE OF clause, UPDATE  
statement [4-22](#)

## FORCE option

error messages [6-23](#)SQL compiler [6-16](#)free function, C language [10-28](#)FREE RESOURCES statement [4-2](#), [4-24](#)**G**GET VERSION statement [6-37](#), [7-7](#)Global memory area [B-2](#)GOTO format, WHENEVER directive [9-8](#)

## Guardian system procedures

See System procedures, Guardian

**H**Help text, similarity check rules [8-12](#)HIGHPIN object-file attribute [7-5](#)HIGHPIN run option, TACL RUN  
command [7-5](#)

## Host object SQL version (HOSV)

C compiler [6-37](#)definition [6-37](#)

## Host variable

colon (:) [1-2](#)creating with INVOKE [2-19](#)data conversion [2-5](#)decimal data type [2-11](#)declaration [1-2](#), [2-1](#)declare sections [2-1](#)definition [2-1](#)DELETE statement [4-13](#)fields in a structure [2-9](#)fixed-point data type [2-11](#)INDICATOR clause [2-6](#)naming conventions [2-2](#)null value [2-17](#)pointer [2-2](#), [2-6](#)SQL cursor [4-19](#), [4-21](#)syntax [2-6](#)TYPE AS clause [2-7](#)VARCHAR data type [2-9](#)Host variable, mismatch effect on SQL  
compilation [6-23](#)Hyphen, double (--) in SQL statements [3-1](#)**I**IN file, SQL compiler [6-14](#)INCLUDE SQLCA directive [5-4](#), [9-12](#)INCLUDE SQLDA directive [10-3](#), [10-24](#)INCLUDE SQLSA directive [9-13](#), [10-24](#)INCLUDE STRUCTURES directive [9-1](#),  
[D-1](#)Index, SQL, changes and program file  
validity [8-3](#)INDICATOR clause with host variable [2-6](#)

## Indicator parameter

function [10-17](#)names buffer [10-18](#)

## Indicator variable

- aggregate function [9-9](#)
- definition [2-1](#)
- host variable [2-6](#)
- INVOKE directive [2-22](#)
- PREFIX and SUFFIX clauses [2-23](#)

ind\_ptr field, initializing [10-29](#)

INFO DEFINE format, EXPLAIN report [6-27](#)

info\_define OSS utility [6-29](#)

Inoperable execution plan [8-10](#)

Input host variable [2-1](#)

Input parameter, dynamic SQL [10-11](#), [10-24](#)

Insert operation of timestamp value [4-10](#)

## INSERT statement

- description [4-8](#)
- null values [2-17](#), [4-9](#)
- scale for numeric data [2-12](#), [2-13](#)

Inspect program, RUND command [7-3](#)

## INTERVAL data types

- description [2-13](#)
- INSERT statement [2-14](#)
- INVOKE directive [2-14](#)

INVALIDATE option, CREATE INDEX statement [8-3](#)

Invalidation caused by DDL operations [8-4](#)

## INVOKE directive

- creating host variables [2-19](#)
- through SQLCI [2-24](#)

## Item codes

- SQLCAGETINFOLIST [5-11](#)
- SQLCAGETINFOLIST parameter [5-10](#)

**J**

JULIANTIMESTAMP procedure [4-10](#)

**K**

Key tags, similarity check rules [8-12](#)

**L**

Library procedures, system [1-4](#)

## List file

- C compiler [6-9](#)
- SQL compiler [6-14](#)

LOAD command and lost open error [4-3](#)

Load time, SQL [8-6](#)

## Local autonomy

- maximizing for distributed database [C-1](#)
- program execution [8-6](#)
- program file validity [8-4](#)
- TACL DEFINES [C-2](#)
- using current statistics [C-2](#)

Local partition, to maximize local autonomy [C-1](#)

LOCKLENGTH attribute, similarity check rules [8-12](#)

Locks, FREE RESOURCES statement [4-24](#)

## Logical DEFINE

See DEFINES, TACL

longlong\_to\_dec C routine [2-11](#)

Loops, infinite, WHENEVER directive [9-8](#)

Lost open error (SQL error -8204) [4-2](#)

**M**

MAXEXTENTS attribute, similarity check rules [8-12](#)

## Maximizing local autonomy

See Local autonomy

Measure program [6-7](#)

MEM option, TACL [B-5](#)

## Memory management

- dynamic allocation [10-18](#)
- estimating use [B-2](#)
- SQLMEM pragma [B-2](#)

Memory model [10-18](#), [10-29](#)

Memory stack overflows [B-4](#)



## Modifying data

DELETE statement [4-12](#), [4-23](#)UPDATE statement [4-10](#)Moving SQL files, effect on SQL validity [8-2](#)

## Multirow operation

DELETE statement [4-13](#)SELECT statement [4-21](#)UPDATE statement [4-12](#)**N**

## Names buffer

determining length [10-7](#)indicator parameters [10-18](#)using with parameter [10-14](#)Naming conventions for host variables [2-2](#)

## Native mode C compiler (NMC)

Guardian environment [6-10](#)SQL pragma [6-7](#)SQLMEM pragma [3-7](#), [B-2](#)NEWPROCESS procedure [B-5](#)NO INVALIDATE option, CREATE INDEX statement [8-3](#)NOEXPLAIN option, SQL compiler [6-16](#)NOFORCE option, SQL compiler [6-16](#)NOOBJECT option, SQL compiler [6-16](#)NOPURGEUNTIL attribute, similarity check rules [8-12](#)NORECOMPILE option, SQL compiler [6-17](#), [8-6](#)NOSQLMAP option, SQL pragma [6-7](#)Not found condition, WHENEVER directive [9-6](#)NOWHENEVERLIST option, SQL pragma [6-7](#)NOXMEM pragma [B-2](#)

## NULL keyword

with INSERT statement [4-9](#)with UPDATE statement [4-12](#)NULL STRUCTURE clause with INVOKE directive [2-23](#)

## Null terminator

C strings [2-7](#)host variables in arrays [2-7](#)

## Null value

definition [2-1](#)dynamic SQL [10-16](#), [10-27](#)input parameters [10-17](#)INSERT statement [2-17](#), [4-9](#)INVOKE directive [2-22](#)names buffer [10-18](#)output variables [10-17](#)parameters [10-17](#)retrieving rows [2-18](#)SELECT statement [2-17](#)testing [2-17](#)UPDATE statement [4-12](#)**O**OBEY command file, format for EXPLAIN report [6-16](#)

## OBEYFORM option

EXPLAIN report [6-27](#)SQL compiler [6-16](#)

## Object file

Accelerator [6-14](#)Binder program [6-11](#)C compiler [6-9](#)SQL compilation [6-12](#), [6-14](#)TACL RUN command [7-3](#)validation [8-1](#)OBJECT option, SQL compiler [6-16](#)

## Object, SQL

access authority for program execution [7-1](#)changes and program file validity [8-3](#)with SQLGETOBJECTVERSION [5-19](#)OPEN procedure [10-36](#)OPEN statement [4-15](#), [4-19](#), [4-22](#), [4-24](#)

## Open System Services (OSS)

- Accelerator [6-31](#)
- Binder program [6-31](#)
- C compilation [6-30](#)
- c89 utility [6-30](#)
- program development [6-28](#)
- shell escape character [6-29](#)
- SQL compiler [6-32](#)
- TACL DEFINES [6-29](#)
- version considerations [6-28](#)

Open tables, SQL [8-7](#)Opening tables and views [4-2](#)Operable execution plan [8-10](#)

## Operating system, Guardian

- SQLCADISPLAY procedure [5-3](#)
- SQLCAFSCODE procedure [5-8](#)
- SQLCATOBUFFER procedure [5-14](#)

## Optimized execution plan

- EXPLAIN PLAN report [6-27](#)
- SQL compiler function [6-13](#)
- statistics requirement [6-23](#)

## OUT file

- C compiler [6-9](#)
- SQL compiler [6-14](#)

Output host variable [2-1](#)

## Output variable

- allocating space [10-33](#)
- displaying [10-33](#)
- dynamic SQL [10-12](#), [10-25](#)

Overflow, stack space [B-4](#)OWNER attribute, similarity check rules [8-12](#)**P**

## Parallel execution plans

- automatic recompilation [8-3](#)
- similarity check [8-9](#)

## PARAM command, TACL

- for SQL program file [7-2](#)

PARAM command, TACL, with SQL compiler [6-22](#)

## Parameter

- dynamic SQL [10-11](#)
- indicator [10-17](#)
- unnamed [10-11](#)
- using a list [10-12](#)
- using in loop [10-13](#)
- value substitution [10-11](#)

## Partition

- local, to maximize local autonomy [C-1](#)
- similarity check rules for attributes [8-12](#)
- skipping unavailable [C-3](#)

Pascal, host language [1-1](#)

## Pathway environment

- dynamic SQL server [10-36](#)
- error checking through requester [4-24](#)
- running C server process [7-6](#)

## Performance

- automatic recompilation [8-5](#)
- INVOKE directive [2-19](#)
- memory considerations [B-4](#)
- SQL cursor considerations [4-2](#)
- SQLSA statistics [9-13](#)

PLAN option for EXPLAIN utility [6-16](#)

## PMSEARCHLIST TACL variable

Pointer, host variable [2-2](#), [2-6](#)

## Pragmas, C compiler

- NOXMEM [B-2](#)
- RUN options [6-9](#)
- RUNNABLE [6-9](#), [6-12](#)
- SQL

- description [3-2](#)

SQLMEM [B-2](#), [B-5](#)SQL, specifying [6-7](#)SYSTYPE [6-9](#)XMEM [B-5](#)XVAR [B-5](#)PREFIX clause, INVOKE statement [2-23](#)

PREPARE statement  
     dynamic SQL compilation [10-25](#)  
 PREPARE statement, SQL compilation errors [6-23](#)  
 Primary key in SELECT statement [4-6](#)  
 Procedures  
     See SQL/MP system procedures  
 process access ID (PAID)  
     DECLARE CURSOR statement [4-18](#)  
     DELETE statement [4-23](#)  
     FETCH statement [4-20](#)  
     OPEN statement [4-19](#)  
     privileges [7-1](#)  
     SELECT statement [4-21](#)  
     SQL cursor requirements [4-16](#)  
     UPDATE statement [4-22](#)  
 Process file segment (PFS) [7-2](#)  
 Process identification numbers (PIN)  
     description [7-4](#)  
     high and low [7-4](#)  
 Processes, concurrent [7-4](#)  
 PROCESS\_CREATE\_ procedure  
     authority [7-1](#)  
     memory [B-5](#)  
     programmatic commands [7-5](#)  
 PROGID attribute [7-1](#)  
 Program catalog version (PCV)  
     CHECK options [6-18](#), [8-10](#)  
     definition [6-37](#)  
     OSS program file [6-33](#)  
     similarity check [8-11](#)  
 Program development, C  
     advantages of INVOKE directive [2-19](#)  
     automatic recompilation dependencies [8-3](#)  
     overview [1-1](#)  
 Program file, SQL  
     binding object file [6-11](#)  
     execution [7-1](#)  
     SQL compilation [6-12](#)

Program file, SQL (continued)  
     TACL DEFINES [7-2](#)  
     TACL RUN command [7-3](#)  
 Program format version (PFV)  
     CHECK options [6-18](#), [8-10](#)  
     definition [6-37](#)  
     OSS program file [6-33](#)  
     similarity check [8-11](#)  
     SQL executor [7-7](#)  
 Program object file  
     See Program file, SQL  
 Program size, estimating [B-2](#)  
 PROGRAMS table  
     file-label and catalog inconsistencies [8-4](#)  
     program invalidation [8-1](#)  
     SIMILARITYINFO column [8-10](#)  
 PROGRAMS table, SQL compilation [6-13](#)  
 Protection view  
     similarity check rules [8-13](#)  
     UPDATE statement [4-10](#)  
 PURGEDATA command and lost open error [4-3](#)

## Q

Question mark (?), unnamed parameter [10-11](#)

## R

READ procedure [1-4](#)  
 READUPDATE procedure [10-36](#)  
 RECEIVE file  
     See \$RECEIVE file  
 RECOMPILE option, SQL compiler [6-17](#), [8-6](#)  
 RECOMPILEALL option, SQL compiler [6-17](#), [8-6](#)  
 RECOMPILEONDEMAND option, SQL compiler [6-17](#)  
 Record descriptions, tables and views [2-19](#)  
 Records, SQLSA statistics [9-14](#)

- Redefinition timestamp
    - program invalidation [8-5](#)
    - similarity check rules [8-12](#)
  - REGISTERONLY clause, SQL compiler [6-17](#)
  - Relational database management system (RDBMS) [1-1](#)
  - Release 1, SQL/MP
    - catalog [5-18](#)
    - object [5-19](#)
    - system software [5-20](#)
  - Release 2, SQL/MP
    - catalog [5-18](#)
    - object [5-19](#)
    - system software [5-20](#)
  - RELEASE1 option in SQL pragma [6-8](#)
  - RELEASE2 option in SQL pragma [6-8](#)
  - RENAME statement, effect on SQL validity [8-3](#)
  - REPLY procedure [10-36](#)
  - Requester, SCREEN COBOL [10-36](#)
  - RESTORE operation and lost open error [4-3](#)
  - RESTORE program with CHECK option [6-18](#)
  - Retrieving SQL data
    - cursor declaration [4-18](#)
    - multiple rows [4-21](#)
    - single row [4-4](#)
  - RISC (TNS/R) system [1-5](#)
  - Row in SQL table
    - DELETE statement [4-12](#), [4-23](#)
    - FETCH statement [4-15](#), [4-20](#)
    - INSERT statement [4-8](#)
    - single-row DELETE statement [4-13](#)
    - single-row SELECT statement [4-4](#)
    - single-row UPDATE statement [4-11](#)
    - SQLSA statistics [9-14](#)
    - UPDATE statement [4-10](#), [4-22](#)
  - RTDU (run time data unit) [6-8](#)
  - RUN command, TACL
    - SQL object file [1-5](#)
    - SQL program file [7-3](#)
  - Run option, TACL
    - C compiler [6-9](#)
    - SQL compiler [6-15](#)
    - SQL program file [7-3](#)
  - Run time data unit (RTDU) [6-8](#)
  - RUND command, TACL [7-3](#)
  - RUNNABLE pragma [6-9](#), [6-12](#)
  - Run-time memory allocation [10-18](#)
  - Run-time recompilation errors [8-9](#)
- ## S
- Sample database [10-37](#), [A-1](#)
  - Sample program
    - basic [10-37](#)
    - detailed [10-42](#)
  - Scale in numeric data
    - INSERT statement [2-12](#), [2-13](#)
    - INVOKE directive [2-13](#)
    - SELECT statement [2-12](#), [2-13](#)
    - SQLDA data\_len field [10-33](#)
    - UPDATE statement [2-12](#), [2-13](#)
  - SCI (SQL compiler interface) [9-3](#)
  - SCREEN COBOL [10-36](#)
  - Section location table (SLT) [6-8](#)
  - SECURE attribute, similarity check rules [8-12](#)
  - Security attribute, effect on SQL validity [8-3](#)
  - SELECT command, Binder [6-12](#)
  - SELECT statement
    - cursor declaration [4-18](#)
    - null values [2-17](#)
    - scale for numeric data [2-12](#), [2-13](#)
    - single row [4-4](#)
  - Semicolon (;) in SQL statements [1-3](#), [3-1](#)
  - SENSITIVE flag, SQL [8-1](#)

## Sequential I/O (SIO) procedures

SQLCADISPLAY [5-3](#)SQLCATOBUFFER [5-14](#)SERIALWRITES attribute, similarity check rules [8-12](#)

## Set operation

automatic recompilation [8-3](#)DELETE statement [4-13](#)UPDATE statement [4-12](#)SETSCALE function [2-5](#), [2-11](#)set\_define OSS utility [6-29](#)show\_define OSS utility [6-29](#)

## Similarity check

ALTER TABLE statement [8-13](#)description [8-9](#)for collations [8-15](#)rules for protection views [8-13](#)rules for tables [8-11](#)SIMILARITYCHECK column, TABLES table [8-11](#)SIMILARITYINFO column, PROGRAMS table [8-10](#)

## Single row in SQL table

SELECT statement [4-4](#)UPDATE statement [4-11](#)

## SIO

See Sequential I/O (SIO)

SKIP UNAVAILABLE PARTITION option, CONTROL TABLE directive [C-3](#)Software Product Revision (SPR) [1-7](#)Sort operations, TACL DEFINES [7-2](#)SORTPROG process [5-3](#), [5-14](#)

SORT\_DEFAULTS DEFINE

See =\_SORT\_DEFAULTS DEFINE

SOURCE directive, SQL [2-2](#)Source file, C compiler [6-9](#)SQL comments, Declare Section [2-2](#)

SQL communications area

See SQLCA structure

SQL compiler interface (SCI) [9-3](#)

## SQL compiler (SQLCOMP)

automatic recompilation [8-5](#)CATALOG clause [6-6](#)DEFINES [6-6](#)description [1-5](#), [6-12](#)determining version [6-36](#)dynamic SQL statements [6-23](#)error messages [6-23](#)EXPLAIN report [6-26](#), [6-27](#)EXPLAIN utility [6-26](#)functions [6-13](#)insufficient information [6-24](#)PARAM command [6-22](#)SQLCOMP command [6-12](#), [6-14](#)unresolved TACL DEFINES [6-24](#)warning messages [6-23](#)SQL Conversational Interface (SQLCI), INVOKE [2-24](#)

SQL descriptor area

See SQLDA structure

SQL directives

See SQL/MP directives

SQL executor, determining version [7-7](#)SQL file attributes [8-3](#)

SQL functions

CONVERTTIMESTAMP [4-10](#)SETSCALE [2-5](#), [2-11](#)

SQL object file

See Program file

SQL object flag [8-1](#)

SQL pragma

description [3-2](#)specifying [10-23](#)SQL pragma, specifying [6-7](#)SQL sensitive flag [8-1](#)

SQL statements

See SQL/MP statements

SQL statistics area

See SQLSA structure

SQL structures, internal [B-1](#)

## SQLCA structure

- automatic SQL recompilation errors [8-9](#)
- description [9-12](#)
- FETCH statement [4-20](#)
- INSERT statement [4-8](#)
- SQLCADISPLAY procedure [5-3](#)
- SQLCAFSCODE procedure [5-8](#)
- SQLCAGETINFOLIST procedure [5-9](#)
- SQLCATOBUFFER procedure [5-14](#)
- UPDATE statement [4-11](#)

## SQLCADISPLAY procedure

- description [5-3](#)
- example [1-4](#)
- SQLCA structure [9-12](#)

## SQLCAFSCODE procedure

- description [5-8](#)
- SQLCA structure [9-12](#)

## SQLCAGETINFOLIST procedure

- description [5-9](#)
- SQLCA structure [9-12](#), [D-1](#)

## SQLCATOBUFFER procedure

- description [5-14](#)
- SQLCA structure [9-12](#)

## SQLCI

See SQL Conversational Interface (SQLCI)

## sqlcode variable

- after DELETE statement [4-13](#)
- after FETCH statement [4-20](#)
- after INSERT statement [4-8](#)
- after UPDATE statement [4-11](#)
- automatic recompilation [8-9](#)
- checking for error [9-4](#)
- data conversion [2-5](#)
- declaration [9-4](#)
- dynamic SQL use [10-23](#)
- WHENEVER directive [9-6](#)

## SQLCOMP command

- description [6-12](#)
- EXPLAIN DEFINES option [7-2](#)

## SQLCOMP command (continued)

- SQLMAP option [9-3](#)
- syntax [6-14](#)

SQLCOMPILE option, RESTORE, CHECK option [6-18](#)

## SQLDA structure

- declarations [10-8](#)
- eye\_catcher field [10-5](#)
- names buffer [10-4](#)
- parameter [10-14](#)
- SQL statements [10-3](#)
- Version 300 template [10-4](#)
- Version 315 (or later) template [10-5](#)
- version management [D-1](#)

SQLDA\_EYE\_CATCHER declaration [10-5](#)SQLDA\_EYE\_CATCHER literal [10-24](#)SQLDA\_HEADER\_LEN declaration [10-5](#)SQLDA\_NAMESBUF\_OVHD\_LEN declaration [10-5](#)SQLDA\_SQLVAR\_LEN declaration [10-5](#)SQLGETCATALOGVERSION procedure [5-18](#)SQLGETOBJECTVERSION procedure [5-19](#)SQLGETSYSTEMVERSION procedure [5-19](#)SQLIN data structure [B-1](#)SQLIVARS data structure [B-1](#)

## SQLMAP option

- SQLCOMP command [9-3](#)

SQLMAP option, SQL pragma [6-7](#)SQLMEM pragma [B-2](#), [B-5](#)

## SQLMSG file

- description [5-2](#)
- file number [5-4](#), [5-15](#)
- SQLCADISPLAY procedure [5-4](#)
- SQLCATOBUFFER procedure [5-15](#)

SQLOVARS data structure [B-1](#)

## SQLSA structure

- declaration [9-13](#)
- description [9-13](#)



## SQLSA structure (continued)

dynamic SQL statement statistics [9-13](#)fields [9-14](#), [9-17](#)INCLUDE SQLSA directive [9-13](#)SQLSADISPLAY procedure [5-20](#)static SQL statement statistics [9-13](#)SQLSADISPLAY procedure [5-20](#)

## SQL/MP database

overview [1-1](#)sample [A-1](#)version management [1-7](#)

## SQL/MP directives

BEGIN DECLARE SECTION [1-2](#), [2-1](#)coding [3-1](#)description [1-3](#)END DECLARE SECTION [1-2](#), [2-1](#)INCLUDE SQLCA [5-4](#), [9-12](#)INCLUDE SQLDA [10-3](#), [10-24](#)INCLUDE SQLSA [9-13](#), [10-24](#)INCLUDE STRUCTURES [9-1](#), [D-1](#)INVOKE [2-19](#)placing in source file [3-2](#)WHENEVER [9-6](#), [10-23](#)

## SQL/MP statements

ADD CONSTRAINT, program  
invalidation [8-5](#)

## ALTER INDEX

error 8204 [4-3](#)program invalidation [8-5](#)

## ALTER TABLE

error 8204 [4-3](#)program invalidation [8-4](#)similarity check [8-13](#)ALTER VIEW [4-3](#)BEGIN WORK [10-27](#)coding guidelines [3-1](#)coding in source file [3-2](#)COMMIT WORK [10-28](#)CREATE CONSTRAINT [4-3](#)

## SQL/MP statements (continued)

## CREATE INDEX

NO INVALIDATE option [8-3](#)program invalidation [8-5](#)DECLARE CURSOR [4-18](#)DELETE [4-12](#), [4-23](#)description [1-3](#)

## DROP CONSTRAINT

error 8204 [4-3](#)program invalidation [8-5](#)

## DROP INDEX

error 8204 [4-3](#)DROP INDEX, program  
invalidation [8-5](#)

## DROP TABLE

error 8204 [4-3](#)program invalidation [8-5](#)

## DROP VIEW

error 8204 [4-3](#)program invalidation [8-5](#)EXECUTE [1-6](#)EXECUTE IMMEDIATE [1-6](#)FETCH [10-12](#)FREE RESOURCES [4-2](#), [4-24](#)GET VERSION [6-37](#), [7-7](#)INSERT [4-8](#)OPEN [4-19](#)placing in source file [3-2](#)PREPARE [10-25](#)RENAME [8-3](#)SELECT [4-4](#)UPDATE [4-10](#)

## UPDATE STATISTICS

error 8204 [4-3](#)execution plans [6-21](#)local autonomy [C-2](#)program invalidation [8-3](#), [8-5](#)

## SQL/MP system procedures

description [5-1](#)

## SQLCADISPLAY

data conversion [2-5](#)example [1-4](#)syntax [5-3](#)SQLCAFSCODE [5-8](#), [9-12](#)SQLCAGETINFOLIST [9-12](#), [D-1](#)SQLCATOBUFFER [9-12](#)SQLGETCATALOGVERSION [5-18](#)SQLGETOBJECTVERSION [5-19](#)SQLGETSYSTEMVERSION [5-19](#)SQLSADISPLAY [5-20](#)Stability, SQL cursor [4-17](#)Stack space requirements [B-4](#)

## Statements

See SQL/MP statements

## Statistics

local autonomy [C-2](#)optimized execution plan [6-23](#)similarity check rules [8-12](#)SQL compilation [6-21](#)SQLCADISPLAY procedure [5-3](#)SQLCAGETINFOLIST procedure [5-9](#)SQLCATOBUFFER procedure [5-14](#)SQLSA structure [9-13](#)SQLSADISPLAY procedure [5-5](#), [5-20](#)UPDATE STATISTICS statement [6-21](#)

## Statistics area, SQL

See SQLSA structure

Status and error reporting [1-5](#), [9-1](#)

## STOREDEFINES option

SQL compiler [6-15](#)SQLCOMP command [6-6](#)STRIP command, Binder [6-12](#)Structure as host variable [2-9](#)SUFFIX clause with INVOKE statement [2-23](#)Swap file volume for SQL compiler [6-22](#)SWAPVOL option, PARAM command [6-22](#)SYNCDEPTH and automatic recompilation [8-3](#)

## System procedures, Guardian

CLOSE [1-4](#)FILE\_GETINFOBYNAME\_ [5-2](#)FILE\_GETINFOLISTBYNAME\_ [5-2](#)FILE\_GETINFOLIST\_ [5-2](#)FILE\_GETINFO\_ [5-2](#)JULIANTIMESTAMP [4-10](#)

## NEWPROCESS

access authority [7-1](#)memory [B-5](#)OPEN [10-36](#)

## PROCESS\_CREATE\_

authority [7-1](#)memory [B-5](#)programmatic commands [7-5](#)READ [1-4](#)READUPDATE [10-36](#)REPLY [10-36](#)WRITEREAD [1-4](#)

## System procedures, SQL/MP

description [5-1](#), [5-2](#)SQLCADISPLAY [1-4](#), [2-5](#), [5-3](#)SQLCAFSCODE [5-8](#)SQLCAGETINFOLIST [5-9](#), [D-1](#)SQLCATOBUFFER [5-14](#)SQLGETCATALOGVERSION [5-18](#)SQLGETOBJECTVERSION [5-19](#)SQLGETSYSTEMVERSION [5-19](#)SQLSADISPLAY [5-20](#)SYSTYPE pragma [6-9](#)

## T

TABLECODE attribute, similarity check rules [8-12](#)TABLES table [8-11](#)

## Table, SQL

changes and program file validity [8-3](#)declaring record descriptions [2-19](#)



## Table, SQL (continued)

maximizing local autonomy for partitions [C-1](#)

open time and automatic SQL recompilation [8-7](#)

SELECT statement [4-4](#)

similarity check rules [8-12](#)

UPDATE statement [4-10](#)

Table, SQL, updating statistics [6-21](#)

## Tandem Advanced Command Language (TACL)

## DEFINES

catalog name [6-6](#), [6-15](#)

CLASS CATALOG [6-6](#), [6-15](#)

description [7-2](#)

maximizing local autonomy [C-2](#)

OSS environment [6-29](#)

RECOMPILE clause [6-17](#)

SQL compilation [6-15](#), [6-24](#)

SQL compiler [6-6](#)

SQL program file [7-2](#)

HIGHPIN run option [7-5](#)PARAM command [7-2](#)

## RUN command

C pragma [3-2](#)

SQL program file [1-5](#), [7-3](#)

RUN command, C compiler [6-9](#)

run options for SQL program file [7-3](#)

RUND command [7-3](#)Tandem NonStop Series/RISC (TNS/R) system [6-11](#), [6-31](#)Terminator, SQL statement [3-1](#)

## Timestamp

check at table open time [8-7](#)

collation check [8-15](#)

INSERT statement [4-10](#)

program validation time [8-1](#)

run-time check [8-7](#)

## TMF

See Transaction Management Facility (TMF)

TNS/R system [1-5](#)

Transaction Application Language (TAL), host language [1-1](#)

Transaction control statements [1-3](#)

## Transaction Management Facility (TMF)

catalog inconsistencies [8-4](#)

data consistency [1-1](#)

example [9-11](#)

FETCH statement [4-20](#)

OPEN statement [4-15](#)

UPDATE statement [4-10](#)

with dynamic SQL [10-27](#)

TRANSIDS tables [6-13](#)

## TYPE AS clause

example [2-14](#)

with date-time data [2-14](#)

with host variable [2-7](#)

with INVOKE directive [2-14](#)

## U

Uncompiled SQL statements, FORCE option [6-23](#)

Underlying SQL tables and similarity check [8-10](#)

Unqualified column names with similarity check [8-14](#)

UPDATE set operations, automatic recompilation [8-3](#)

## UPDATE statement

description [4-10](#)

multiple rows [4-12](#)

null values [4-12](#)

scale for numeric data [2-12](#), [2-13](#)

set of rows [4-12](#)

single row [4-11](#)

using parameter [10-12](#)

UPDATE STATISTICS statement  
     effect on program invalidation [8-3](#)  
     effect on SQL validity [8-3](#)  
     error 8204 [4-3](#)  
     maximizing local autonomy [C-2](#)  
     RECOMPILE option, program  
         invalidation [8-5](#)  
 UPDATE STATISTICS statement, SQL  
 compiler [6-21](#)  
 UPDATE WHERE CURRENT clause for a  
 cursor [10-21](#)  
 USAGES table  
     CHECK INOPERABLE PLANS  
     option [8-5](#)  
     SQL compiler access [6-13](#)  
     SQL compiler entries [8-1](#)  
     unrecorded program  
     dependencies [6-23](#)  
 USER option, SQLMEM pragma [B-2](#)  
 USING DESCRIPTOR clause  
     FETCH statement [10-12](#)  
     for a cursor [10-21](#)

## V

VALID flag, SQL  
     checking [8-1](#)  
     PROGRAMS table [8-4](#)  
 Validation, program file  
     checking [8-1](#)  
 Validation, program file, FORCE  
 option [6-23](#)  
 VARCHAR data type  
     correspondence in C [2-3](#), [2-4](#)  
     host variable declaration [2-9](#)  
 Variable-length character data, host  
 variable declaration [2-9](#)  
 VERIFIEDWRITES attribute, similarity  
 check rules [8-12](#)  
 VERIFY utility, SQL [8-1](#)  
 Version management  
     C compiler [6-36](#), [9-3](#)

Version management (continued)  
     description [1-7](#)  
     displaying information  
         SQLGETCATALOGVERSION  
         procedure [5-18](#)  
         SQLGETOBJECTVERSION  
         procedure [5-19](#)  
         SQLGETSYSTEMVERSION  
         procedure [5-19](#)  
     INCLUDE STRUCTURES directive [9-1](#)  
     SQL compiler [6-36](#)  
     SQL Executor [7-7](#)  
     SQL program file [6-37](#)  
     VPROC program [9-3](#)  
 vi text editor [6-28](#)  
 View, SQL  
     changes and program file validity [8-3](#)  
     declaring record descriptions [2-19](#)  
 Virtual sequential block buffering (VSBB),  
 SQL cursor operations [4-17](#)  
 VPROC program [9-3](#)

## W

Warning messages  
     detecting with WHENEVER  
     directive [9-6](#)  
 Warning messages, SQL compiler [6-23](#)  
 WHENEVER directive  
     description [9-6](#)  
     disabling checking [9-7](#)  
     dynamic SQL [10-23](#)  
     enabling checking [9-7](#)  
     scope [9-7](#)  
 WHENEVERLIST option, SQL pragma [6-7](#)  
 WHERE CURRENT OF clause, UPDATE  
 statement [4-22](#)  
 WRITEREAD procedure [1-4](#)

## X

XMEM pragma [10-18](#), [B-5](#)

XVAR pragma [B-5](#)

## Z

ZZBlnnnn object file [6-9](#)

## Special Characters

" (double quotes) in SQL statements [3-1](#)

#define C directive [2-1](#)

#include C directive [2-2](#), [11-2](#)

\$0 collector process [B-5](#)

\$RECEIVE file [10-36](#)

\$SYSTEM.SYSTEM.SQLMSG file [5-2](#)

\* (asterisk)

    with pointer as host variable [2-6](#)

    with similarity check [8-13](#)

-Wverbose flag and EXPLAIN utility [6-33](#)

-- (double hyphen) in SQL statements [3-1](#)

/bin/compilers directory [6-30](#)

/nonnative/bin/compilers directory [6-30](#)

: (colon) with host variable [1-2](#), [2-6](#)

; (semicolon) in SQL statements [1-3](#), [3-1](#)

=\_DEFAULTS DEFINE, TACL [6-27](#)

=\_SORT\_DEFAULTS DEFINE [7-2](#)

? (question mark), unnamed  
parameter [10-11](#)

\ (backslash), OSS shell escape  
character [6-29](#)