

HP NonStop SQL/MX Release 3.2.1 Programming Manual for C and COBOL

Abstract

This manual explains how to use embedded SQL for HP NonStop™ SQL/MX for C, C++, and COBOL. In NonStop SQL/MX, a C, C++, or COBOL program uses embedded SQL/MX statements to access HP NonStop SQL/MP and SQL/MX databases.

Product Version

NonStop SQL/MX Release 3.2.1

Supported Release Version Updates (RVUs)

This publication supports J06.14 and all subsequent J-series RVUs and H06.25 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
663854-005	June 2013

Document History

Part Number	Product Version	Published
544617-002	NonStop SQL/MX Release 2.3	February 2010
544617-003	NonStop SQL/MX Release 2.3	August 2010
663854-001	NonStop SQL/MX Release 3.1	October 2011
663854-002	NonStop SQL/MX Release 3.2	August 2012
663854-004	NonStop SQL/MX Release 3.2.1	February 2013
663854-005	NonStop SQL/MX Release 3.2.1	June 2013

Legal Notices

© Copyright 2013 Hewlett-Packard Development Company L.P.

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, Itanium, 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. This documentation and the 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.

This 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.

Printed in the US

HP NonStop SQL/MX Release

3.2.1 Programming Manual for C and COBOL

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

[Legal Notices](#)

[What's New in This Manual](#) xv[Manual Information](#) xv[New and Changed Information](#) xv[About This Manual](#) xix[Audience](#) xix[Organization](#) xix[Related Documentation](#) xx[Examples in This Manual](#) xxiii[Notation Conventions](#) xxiii[Change Bar Notation](#) xxvi[HP Encourages Your Comments](#) xxvi

[1. Introduction](#)

[Referencing Database Object Names](#) 1-1[NonStop SQL/MX Release 2.x](#) 1-1[NonStop SQL/MX Release 1.x](#) 1-1[Embedding SQL Statements](#) 1-2[Embedding SQL Statements in DLL](#) 1-2[Declaring and Using Host Variables](#) 1-3[Declaring Host Variables](#) 1-3[Using Host Variables](#) 1-4[Using DML Statements to Manipulate Data](#) 1-4[Declaring and Using Static SQL Cursors](#) 1-5[Using Dynamic SQL](#) 1-7[Using Descriptor Areas for Dynamic SQL](#) 1-7[Using Dynamic SQL Cursors](#) 1-8[Using DML Statements With Rowsets](#) 1-8[Improving Performance by Using Rowsets](#) 1-8[Declaring a Rowset](#) 1-9[Using a Rowset in a Query](#) 1-9

Processing Exception Conditions	1-10
Checking SQLSTATE	1-10
Using WHENEVER	1-10
Using GET DIAGNOSTICS	1-11
Ensuring Data Consistency	1-11
Compiling and Building an Application	1-12
Processing With Embedded Module Definitions	1-12
Processing With Module Definition Files	1-12
General Instructions	1-13
SQL/MX Host Language Preprocessor	1-13
SQL/MX Compiler	1-14
Host Language Compiler	1-14
Program and Module Management	1-14

[2. Embedded SQL Statements](#)

Syntax for Coding SQL Statements	2-1
Guidelines for Coding SQL Statements	2-1
Placement of SQL Statements	2-2
MODULE Directive	2-2
Host Variable Declarations	2-2
Nonexecutable SQL Statements	2-4
Executable SQL Statements	2-4
Embedded SQL Declarations and Statements	2-6
Considerations for Embedding DDL and DML Statements	2-12
Considerations for Embedding the UPDATE STATISTICS Statement	2-12
Using CONTROL Statements	2-12
ANSI Compliance and Portability	2-13
Static and Dynamic CONTROL Statements	2-13
CONTROL, Line Order Scope, and Static SQL programs	2-13
CONTROL, Flow Control Scope, and Dynamic SQL programs	2-13

[3. Host Variables in C/C++ Programs](#)

Specifying a Declare Section	3-1
C Host Variable Data Types	3-2
Character Host Variables	3-3
Date-Time and Interval Host Variables	3-4
Numeric Host Variables	3-6
Floating-Point Host Variables	3-7
Using Corresponding SQL and C Data Types	3-8
Extended Host Variable Data Types and Generated C Data Types	3-11

Data Conversion	3-13
Specifying Host Variables in SQL Statements	3-15
Using Host Variables in a C/C++ Program	3-16
Character Set Data	3-16
Fixed-Length Character Data	3-17
Variable-Length Character Data	3-19
Numeric Data	3-22
Date-Time and Interval Data	3-33
Host Variables in C Structures	3-39
Host Variables as Data Members of a C++ Class	3-39
Using Indicator Variables in a C/C++ Program	3-40
Inserting Null	3-40
Testing for Null or a Truncated Value	3-41
Retrieving Rows With Nulls	3-41
Creating C Host Variables Using INVOKE	3-42
Using the INVOKE Directive	3-42
INVOKE and Date-Time and Interval Host Variables (SQL/MX Release 1.8 Applications)	3-43
INVOKE and Floating-Point Host Variables	3-44
C Data Types Generated by INVOKE	3-45
Using Indicator Variables With the INVOKE Directive	3-48
C Example of Using INVOKE	3-50
Character Set Examples	3-51
Selecting From a UCS2 Character Set Into a VARCHAR Host Variable	3-52
Fetching From a UCS2 Character Set into a VARCHAR Host Variable	3-52
Selecting From an ISO88591 Character Set Into a UCS2 Host Variable	3-53

4. Host Variables in COBOL Programs

Specifying a Declare Section	4-1
COBOL Host Variable Data Types	4-2
Using Corresponding SQL and COBOL Data Types	4-5
Data Conversion	4-8
Specifying Host Variables in SQL Statements	4-9
Using Host Variables in a COBOL Program	4-10
Character Set Data	4-10
Fixed-Length Character Data	4-11
Variable-Length Character Data	4-12
Numeric Data	4-12
Date-Time and Interval Data	4-13

Using COBOL Data Description Clauses	4-18
Using Indicator Variables in a COBOL Program	4-19
Inserting Null	4-19
Testing for Null or a Truncated Value	4-20
Retrieving Rows With Nulls	4-21
Creating COBOL Host Variables Using INVOKE	4-22
Using the INVOKE Directive	4-23
INVOKE and Date-Time and Interval Host Variables (SQL/MX Release 1.8 Applications)	4-23
COBOL Record Descriptions Generated by INVOKE	4-23
Using Indicator Variables With the INVOKE Directive	4-27
COBOL Example of Using INVOKE	4-29
Character Set Examples	4-30
Selecting From a UCS2 Character Set Into a VARCHAR Host Variable	4-31
Fetching From a UCS2 Character Set into a VARCHAR Host Variable	4-31

5. Simple and Compound Statements

Single-Row SELECT Statement	5-2
Using a Primary Key Value to Select Data	5-2
Selecting a Column With Date-Time or INTERVAL Data Type	5-3
INSERT Statement	5-4
Inserting Rows	5-5
Inserting Null	5-6
Inserting a Date-Time Value	5-7
Inserting an Interval Value	5-8
Searched UPDATE Statement	5-9
Updating a Single Row	5-10
Updating Multiple Rows	5-11
Updating Columns To Null	5-11
Searched DELETE Statement	5-12
Deleting a Single Row	5-12
Deleting Multiple Rows	5-12
Compound Statements	5-13
Assignment Statement	5-15
IF Statement	5-16
Using PROTOTYPE Host Variables as Table Names	5-17

6. Static SQL Cursors

DML Statements for Static SQL Cursors	6-1
Steps for Using a Static SQL Cursor	6-2

Declare Required Host Variables	6-4
Declare the Cursor	6-4
Initialize the Host Variables	6-5
Open the Cursor	6-5
Retrieve the Values	6-6
Process the Retrieved Values	6-7
Fetch the Next Row	6-10
Close the Cursor	6-11
Using Date-Time and INTERVAL Data Types	6-12
Standard Date-Time Example	6-12
Nonstandard SQL/MP DATETIME Example	6-13
Interval Example	6-13
Using Floating-Point Data Types	6-14
Considerations When Using a Cursor	6-14
Cursor Position	6-15
Cursor Stability	6-15
Cursor Sensitivity	6-16

7. Static Rowsets

What Are Rowsets?	7-1
Using Rowsets	7-2
Declaring Host Variable Arrays as Rowsets	7-2
Rowset Host Variable Pointers	7-3
Considerations for Rowset Size	7-4
Specifying Rowset Arrays	7-4
Using Rowset Arrays for Input	7-6
Using Rowset Arrays for Output	7-6
Using Rowset Arrays in DML Statements	7-7
Selecting Rows Into Rowset Arrays	7-7
Selecting Rowsets With a Cursor	7-16
Inserting Rows From Rowset Arrays	7-18
Updating Rows by Using Rowset Arrays	7-21
Deleting Rows by Using Rowset Arrays	7-23
Specifying Size and Row ID for Rowset Arrays	7-24
Limiting the Size of the Input Rowset	7-26
Limiting the Size of the Input Rowset When Declaring a Cursor	7-27
Limiting the Size of the Output Rowset	7-28
Using the Index Identifier	7-29
Specifying Rowset-Derived Tables	7-32

Using Rowset-Derived Tables in DML Statements	7-33
Selecting From Rowset-Derived Tables	7-33
Selecting From Rowset-Derived Tables With a Cursor	7-36
Inserting Rows From Rowset-Derived Tables	7-38
Limiting the Size of a Rowset-Derived Table	7-39
Inserting Null	7-40
Updating Rows by Using Rowset-Derived Tables	7-41
Deleting Rows by Using Rowset-Derived Tables	7-43

8. Name Resolution, Similarity Checks, and Automatic Recompilation

Name Resolution	8-1
Table and View Name References	8-1
Precedence of Object Name Qualification	8-5
Compile-Time Name Resolution for SQL/MP Objects	8-6
Late Name Resolution	8-6
Distributed Database Considerations	8-8
RDF Considerations	8-8
Similarity Checks and Automatic Recompilation	8-9
Similarity Check	8-9
Automatic Recompilation	8-18
Recommended Recompilation Settings for OLTP Programs	8-19

9. Dynamic SQL

Statements for Dynamic SQL With Arguments	9-2
Input Parameters and Output Variables	9-2
Floating-Point Variables	9-2
Steps for Using Dynamic SQL With Argument Lists	9-3
Declare a Host Variable for the Dynamic SQL Statement	9-4
Move the Statement Into the Host Variable	9-5
Prepare the SQL Statement	9-5
Set Explicit Input Values	9-6
Execute the Prepared Statement	9-6
Deallocate the Prepared Statement	9-7
Using EXECUTE IMMEDIATE	9-7
Setting Default Values Dynamically	9-8

10. Dynamic SQL With Descriptor Areas

Statements for Dynamic SQL With Descriptors	10-1
SQL Descriptor Areas	10-2

SQL Item Descriptors	10-2
Allocating an SQL Descriptor Area	10-3
Deallocating an SQL Descriptor Area	10-3
Input Parameters	10-3
Describing Input Parameters	10-4
Setting the Data Values of Input Parameters	10-4
Setting Input Parameter Information Without DESCRIBE INPUT	10-6
Output Variables	10-7
Describing Output Variables	10-7
Getting the Values of Output Variables	10-7
Consideration—Retrieving Multiple Values From a Large Buffer	10-8
Steps for Using SQL Item Descriptor Areas	10-12
Declare a Host Variable for the Dynamic SQL Statement	10-14
Construct the SQL Statement From User Input	10-14
Allocate Input and Output SQL Descriptor Areas	10-14
Prepare the SQL Statement	10-15
Describe the Input Parameters and the Output Variables	10-15
Set Explicit Input Values	10-16
Execute the Prepared Statement	10-18
Get the Count and Descriptions of the Output Variables	10-19
Deallocate the Prepared Statement and the SQL Descriptor Areas	10-20
Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data	10-21
Using SQL Descriptor Areas to Retrieve ISO88591 Data to UCS2 Host Variables	10-21

11. Dynamic SQL Cursors

Statements for Dynamic SQL Cursors	11-1
Steps for Using a Dynamic SQL Cursor	11-2
Declare Required Host Variables	11-4
Prepare the Cursor Specification	11-4
Declare the Cursor	11-4
Initialize the Dynamic Input Parameters	11-5
Open the Cursor	11-5
Retrieve the Values	11-5
Process the Retrieved Values	11-6
Fetch the Next Row	11-6
Close the Cursor and Deallocate the Prepared Statement	11-6
Using Date-Time and INTERVAL Data Types	11-7
Standard Date-Time Example	11-7

[Interval Example](#) 11-8

[Nonstandard SQL/MP DATETIME Example](#) 11-9

[Dynamic SQL Cursors Using Descriptor Areas](#) 11-10

12. Dynamic SQL Rowsets

[Using Dynamic SQL Rowsets](#) 12-1

[Preparing an SQL Statement With Dynamic Rowsets](#) 12-2

[Specification of an Rowset Parameter in the PREPARE String](#) 12-3

[Matching Compile-Time Specified Length With Execution-Time Length](#) 12-4

[Dynamic SQL With Argument Lists](#) 12-5

[Using the SET DESCRIPTOR Statement](#) 12-5

[Setting the Rowset-Specific Descriptor Fields](#) 12-5

[Exclusive Use of VARIABLE_POINTER and INDICATOR_POINTER](#) 12-9

[Using the GET DESCRIPTOR Statement](#) 12-9

[Using the DESCRIBE INPUT Statement](#) 12-10

13. Exception Handling and Error Conditions

[Checking the SQLSTATE Variable](#) 13-1

[Declaring SQLSTATE](#) 13-2

[SQL:1999 SQLSTATE Values](#) 13-2

[SQL/MX SQLSTATE Values](#) 13-3

[Using SQLSTATE](#) 13-4

[Checking the SQLCODE Variable](#) 13-5

[Declaring SQLCODE](#) 13-5

[Declaring SQLCODE and SQLSTATE](#) 13-5

[SQLCODE Values](#) 13-10

[Using SQLCODE](#) 13-10

[SQL/MX Exception Condition Messages](#) 13-11

[Viewing the SQL Messages](#) 13-12

[Accessing SQL Messages Within a Program](#) 13-12

[Using the WHENEVER Statement](#) 13-13

[Precedence of Multiple WHENEVER Declarations](#) 13-14

[Determining the Scope of a WHENEVER Declaration](#) 13-14

[Enabling and Disabling the WHENEVER Declaration](#) 13-14

[Saving and Restoring SQLSTATE or SQLCODE](#) 13-15

[Declaring SQLSTATE or SQLCODE in an Error Routine](#) 13-16

[Accessing and Using the Diagnostics Area](#) 13-17

[Using the GET DIAGNOSTICS Statement](#) 13-18

[Getting Statement and Condition Items](#) 13-18

[Special SQL/MX Error Conditions](#) 13-20

Lost Open Error (8574)	13-20
Occurrences of the Lost Open Error	13-20
Recovering From the Lost Open Error	13-21

14. Transaction Management

Transaction Control Statements	14-1
Steps for Ensuring Data Consistency	14-1
Declaring Required Variables	14-3
Setting Attributes for Transactions	14-3
Starting a Transaction	14-6
Processing Database Changes	14-7
Testing for Errors	14-7
Committing Database Changes if No Errors Occur	14-8
Undoing Database Changes if an Error Occurs	14-8

15. C/C++ Program Compilation

Compiling SQL/MX Applications and Modules	15-2
Compiling Embedded SQL C/C++ Programs With Embedded Module Definitions	15-2
Compiling Embedded SQL C/C++ Programs With Module Definition Files	15-6
Creating Modules: From Development to Production	15-8
Running the SQL/MX C/C++ Preprocessor	15-8
Preprocessor Functions	15-9
Preprocessor Output	15-17
OSS-Hosted SQL/MX C/C++ Preprocessor	15-19
Windows-Hosted SQL/MX C/C++ Preprocessor	15-26
Running the C/C++ Compiler and Linker	15-34
Running the SQL/MX Compiler	15-36
Compiling Embedded Module Definitions	15-37
MXCMP Environment Variable	15-41
MXCMPUM Environment Variable	15-41
Compiling a Module Definition File	15-42
c89 Utility: Using One Command for All Compilation Steps	15-44
c89 Examples With Embedded Module Definitions	15-49
c89 Examples With Module Definition Files	15-52
Examples of Building and Deploying Embedded SQL C/C++ Programs	15-55
Building a C/C++ Program With Embedded SQL Statements on Windows	15-55
Developing a Native C/C++ Program With Embedded SQL/MX Statements on OSS	15-57

Building and Deploying a C Application With Embedded Module Definitions and Module Definition Files	15-58
Quick Builds and mxcmp Defaults in a One-File Application Deployment	15-60
Deploying a Static SQL Application to an RDF System	15-62
Building SQL/MX C/C++ Applications to Run in the Guardian Environment	15-66
Building SQL/MX Guardian Applications in the Guardian Environment	15-67
Building SQL/MX Guardian Applications in the OSS Environment	15-72
Running an SQL/MX Application	15-72
Running the SQL/MX Program File	15-73
Understanding and Avoiding Some Common Run-Time Errors	15-73
Debugging a Program	15-75
Displaying Query Execution Plans	15-75

16. COBOL Program Compilation

Compiling SQL/MX Applications and Modules	16-2
Compiling Embedded SQL COBOL Programs With Embedded Module Definitions	16-3
Compiling Embedded SQL COBOL Programs With Module Definition Files	16-6
Creating Modules: From Development to Production	16-8
Running the SQL/MX COBOL Preprocessor	16-9
Preprocessor Functions	16-9
Preprocessor Output	16-11
OSS-Hosted SQL/MX COBOL Preprocessor	16-13
Windows-Hosted SQL/MX COBOL Preprocessor	16-18
Running the COBOL Compiler and Linker	16-23
Running the SQL/MX Compiler	16-25
Compiling Embedded Module Definitions	16-25
MXCMP Environment Variable	16-30
MXCMPUM Environment Variable	16-30
Compiling a Module Definition File	16-30
ecobol or nmcobol Utility: Using One Command for All Compilation Steps	16-33
ecobol and nmcobol Examples With Embedded Module Definitions	16-41
ecobol and nmcobol Examples With Module Definition Files	16-44
Combining Embedded Module Definitions and Module Definition Files	16-46
Building SQL/MX COBOL Applications to Run in the Guardian Environment	16-47
Building SQL/MX Guardian Applications in the Guardian Environment	16-47
Building SQL/MX Guardian Applications in the OSS Environment	16-50
Running an SQL/MX Application	16-51
Running the SQL/MX Program File	16-52
Understanding and Avoiding Common Run-Time Errors	16-52

[Displaying Query Execution Plans](#) 16-55

[17. Program and Module Management](#)

[Program Files](#) 17-1

[Managing Program Files](#) 17-3

[Generating Locally or Globally Placed Modules](#) 17-3

[Managing the Coexistence of Globally and Locally Placed Modules](#) 17-4

[Generating modules in a user-specified location](#) 17-6

[Specifying the search locations for the module files](#) 17-7

[Managing Modules](#) 17-8

[Module Management Behavior](#) 17-8

[Influencing Module Management Behavior](#) 17-9

[Module Management Naming](#) 17-9

[How Modules Are Named](#) 17-10

[Effect of Module Management Naming](#) 17-13

[Specifying the search locations of the module files](#) 17-13

[Targeting](#) 17-14

[Effect of the Target Attribute](#) 17-15

[Targeting Example for C: Using ModuleTableSet \(MTSS\)](#) 17-15

[Targeting Example for C: Using Build Subdirectory](#) 17-17

[Targeting Example for COBOL: Using ModuleTableSet \(MTSS\)](#) 17-18

[Targeting Example for COBOL: Using a Build Subdirectory](#) 17-20

[Versioning](#) 17-21

[Grouping](#) 17-23

[A. C Sample Programs](#)

[Using a Static SQL Cursor](#) A-1

[Ensuring Data Consistency](#) A-4

[Using Argument Lists in Dynamic SQL](#) A-5

[Using SQL Descriptor Areas in Dynamic SQL](#) A-7

[Using SQL Descriptor Areas With DESCRIBE](#) A-7

[Using SQL Descriptor Areas Without DESCRIBE](#) A-12

[Using a Dynamic SQL Cursor](#) A-15

[Using a Dynamic SQL Cursor](#) A-15

[Using a Dynamic SQL Cursor With Descriptor Area](#) A-17

[Using a Dynamic SQL Rowset](#) A-26

[Using SQL Descriptors to Select KANJI and KSC5601 Data](#) A-28

[DDL for KANJI and KSC4501 Table Columns](#) A-28

[Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data](#) A-29

[Using SQL Descriptors to Select UCS2 Data](#) A-35

B. C++ Sample Program

[Ensuring Data Consistency](#) B-1

C. COBOL Sample Programs

[Using a Static SQL Cursor](#) C-1

[Ensuring Data Consistency](#) C-4

[Using Argument Lists in Dynamic SQL](#) C-6

[Using SQL Descriptor Areas in Dynamic SQL](#) C-9

[Using a Dynamic SQL Cursor](#) C-13

Index

Examples

Example 2-1.	Static and Dynamic SQL and CONTROL Scope	2-14
Example 3-1.	CREATE TABLE Statement	3-47
Example 3-2.	C Structure Generated by INVOKE	3-48
Example 3-3.	C INVOKE	3-51
Example 4-1.	Null Test Example	4-21
Example 4-2.	CREATE TABLE Statement	4-24
Example 4-3.	COBOL Record Description Generated by INVOKE	4-25
Example 4-4.	INVOKE Example	4-30
Example 10-1.	C VARIABLE_POINTER Example	10-9
Example A-1.	Using a Static SQL Cursor	A-1
Example A-2.	Using TMF to Ensure Data Consistency	A-4
Example A-3.	Using Argument Lists in Dynamic SQL	A-5
Example A-4.	Using SQL Descriptor Areas With DESCRIBE	A-8
Example A-5.	Using SQL Descriptor Areas Without DESCRIBE	A-12
Example A-6.	Using a Dynamic SQL Cursor	A-15
Example A-7.	Using a Dynamic SQL Cursor With Descriptor Areas	A-18
Example A-8.	Dynamic SQL Rowsets	A-26
Example A-9.	DDL for KANJI and KSC4501 Table Columns	A-28
Example A-10.	Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data	A-30
Example A-11.	Using SQL Descriptors to Select UCS2 Data	A-35
Example B-1.	Using TMF to Ensure Data Consistency	B-1
Example C-1.	Using a Static SQL Cursor	C-1
Example C-2.	Using TMF to Ensure Data Consistency	C-4
Example C-3.	Using Argument Lists in Dynamic SQL	C-6
Example C-4.	Using Descriptor Areas With DESCRIBE	C-9
Example C-5.	Using a Dynamic SQL Cursor	C-13

Figures

Figure 6-1.	Using a Static SQL Cursor in a C Program	6-2
Figure 6-2.	Using a Static SQL Cursor in a COBOL Program	6-3
Figure 9-1.	Using Dynamic SQL in a C Program	9-3
Figure 9-2.	Using Dynamic SQL in a COBOL Program	9-4
Figure 10-1.	Using SQL Descriptor Areas in a C Program	10-12
Figure 10-2.	Using SQL Descriptor Areas in a COBOL Program	10-13
Figure 11-1.	Using a Dynamic SQL Cursor in a C Program	11-2
Figure 11-2.	Using a Dynamic SQL Cursor in a COBOL Program	11-3
Figure 14-1.	Coding Transaction Control Statements in a C Program	14-1
Figure 14-2.	Coding Transaction Control Statements in a COBOL Program	14-2
Figure 15-1.	Compiling Embedded SQL C/C++ Programs With Embedded Module Definitions	15-3
Figure 15-2.	Compiling Embedded SQL C/C++ Programs With Module Definition Files	15-6
Figure 15-3.	c89 Generating Annotated Source With Embedded Module Definitions	15-49
Figure 15-4.	c89 Generating Module Definition Files	15-52
Figure 16-1.	Compiling Embedded SQL COBOL Programs With Embedded Module Definitions	16-3
Figure 16-2.	Compiling Embedded SQL COBOL Programs With Module Definition Files	16-6
Figure 16-3.	ecobol or nmcobol Generating Annotated Source With Embedded Module Definitions	16-41
Figure 16-4.	ecobol or nmcobol Generating Module Definition Files	16-44
Figure 17-1.	Module Name Length	17-12

Tables

Table 2-1.	MODULE Directive	2-6
Table 2-2.	Embedded SQL Statements in SQL Declare Section	2-6
Table 2-3.	Nonexecutable SQL Statements	2-7
Table 2-4.	Executable SQL Statements	2-8
Table 3-1.	Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, and LARGEINT Data Types	3-9
Table 3-2.	Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for Float Data Types	3-10
Table 3-3.	Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for Date-Time Data Types	3-11
Table 3-4.	Corresponding SQL, C Host Variable Data Types, and Translated C Declarations	3-12

Table 3-5.	Host Variable Usage for NUMERIC or PICTURE 9's COMP Data	3-31
Table 3-6.	Host Variable Usage for DECIMAL or PICTURE 9's DISPLAY Data	3-32
Table 3-7.	Lengths of C Target Arrays for TIME and TIMESTAMP	3-35
Table 3-8.	INVOKE and Floating-Point Host Variables	3-45
Table 4-1.	COBOL Character Host Variables and Their SQL Data Type Equivalents and COBOL Translations	4-5
Table 4-2.	Corresponding SQL, COBOL Host Variable Data Types, and Translated COBOL Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, LARGEINT, and Date-Time Data Types	4-7
Table 4-3.	Lengths of Target Arrays for TIME and TIMESTAMP	4-14
Table 4-4.	Interpretation of COBOL Data Description Clauses	4-18
Table 4-5.	Changes Made by INVOKE in Generated Host Variables	4-27
Table 12-1.	Minimum Values for ROWSET_VAR_LAYOUT_SIZE Descriptor Field	12-7
Table 13-1.	SQL:1999 SQLSTATE Class and Subclass Values	13-2
Table 13-2.	Mapping of SQLCODE to SQL/MX-Defined SQLSTATE Values	13-4
Table 13-3.	SQLCODE and SQLSTATE missing declaration	13-6
Table 13-4.	SQLCODE and SQLSTATE incorrect declaration	13-7
Table 13-5.	SQLCODE Values	13-10
Table 15-1.	HP NonStop C/C++ Compilers for Embedded SQL/MX Programs	15-35
Table 15-2.	Module Schemas and Export Files for RDF SQL Application Deployment Example	15-62
Table 16-1.	HP NonStop COBOL Compilers for Embedded SQL/MX Programs	16-24
Table 17-1.	File Naming Conventions	17-1
Table 17-2.	Preprocessor Interpretation of SQLMX_PREPROCESSOR_VERSION Environment Variable and -m and -x Options	17-9
Table 17-3.	Module Management Naming	17-13

What's New in This Manual

Manual Information

Abstract

This manual explains how to use embedded SQL for HP NonStop™ SQL/MX for C, C++, and COBOL. In NonStop SQL/MX, a C, C++, or COBOL program uses embedded SQL/MX statements to access HP NonStop SQL/MP and SQL/MX databases.

Product Version

NonStop SQL/MX Release 3.2.1

Supported Release Version Updates (RVUs)

This publication supports J06.14 and all subsequent J-series RVUs and H06.25 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
663854-005	June 2013

Document History

Part Number	Product Version	Published
544617-002	NonStop SQL/MX Release 2.3	February 2010
544617-003	NonStop SQL/MX Release 2.3	August 2010
663854-001	NonStop SQL/MX Release 3.1	October 2011
663854-002	NonStop SQL/MX Release 3.2	August 2012
663854-004	NonStop SQL/MX Release 3.2.1	February 2013
663854-005	NonStop SQL/MX Release 3.2.1	June 2013

New and Changed Information

Changes to the 663854-005 manual:

Removed a note about 64-bit application support in Chapter 9.

Changes to the 663854-004 manual:

- Updated the section [Specifying a Declare Section](#) on page 3-1.
- Updated the section [C Host Variable Data Types](#) on page 3-2.
- Updated the section, [Numeric Host Variables](#) on page 3-6.

- Updated the table [Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, and LARGEINT Data Types](#) on page 3-9.
- Added the section [Host Variable Pointers](#) on page 3-14.
- Updated the table [Host Variable Usage for NUMERIC or PICTURE 9's COMP Data](#) on page 3-31.
- Updated the example in the section [Host Variables in C Structures](#) on page 3-40.
- Added the section [Rowset Host Variable Pointers](#) on page 7-4.
- Updated the section, [Similarity Check Criteria](#) on page 8-11.
- Updated the section [C #include directive](#) on page 15-10.
- Updated the option [-O](#) on page 15-25.
- Updated the option [\[-O\]](#) on page 16-17.

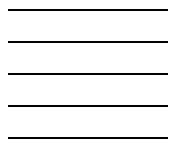
Changes to the 663854-002 manual:

- Updated the section [Numeric Host Variables](#) on page 3-6.
- Updated the table [Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, and LARGEINT Data Types](#) on page 3-9.
- Added a new section, [Initializing BigNum Data Types](#) on page 3-24.
- Added a new section, [Considerations for BigNum Arithmetic function](#) on page 3-27.
- Added a new section, [GNU GMP library for BigNum](#) on page 3-28
- Added a new section, [BigNum Format for TMFARLIB](#) on page 3-30.
- Added a new section [Retrieving the Row Number for a Failed Operation](#) on page 7-10.
- Added a new section [Late Name Resolution for Tables Referred by the View](#) on page 8-7.
- Updated the section [Similarity Check](#) on page 8-9.
- Added a new section [Similarity Check Criteria for a View](#) on page 8-13.
- Updated the syntax and its description in section [Syntax for the OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-20.
- Added new example to the section [Example—mxsqlc](#) on page 15-25.
- Updated the syntax and its description in section [Syntax for the Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-28.
- Added new example to the section [Example—mxsqlc](#) on page 15-34.

- Updated the contents in [Compiling Embedded Module Definitions](#) on page 15-37.
- Added 64-bit examples to the section c89 Examples With Module Definition Files on page [15-53](#).
- Updated the contents in [Compiling Embedded Module Definitions](#) on page 16-25.
- Added -Wsqlconnect compiler option in [-Wsqlconnect](#) on page 16-38.
- Added -HP_NSK_CONNECT_MODE environment variable option in [HP_NSK_CONNECT_MODE](#) on page 16-39.

Changes to the 663854-001 manual:

- Updated the contents in [Embedding SQL Statements in DLL](#) on page 1-2.
- Updated the contents in [Using the VARCHAR compatible structure to hold VARCHAR data](#) on page 3-20.
- Updated the contents in [Declaring SQLSTATE](#) on page 13-2.
- Updated the contents in [Checking the SQLCODE Variable](#) on page 13-5.
- Updated the contents in [Preprocessor Functions](#) on page 15-9.
- Updated the contents in [OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-19
- Updated the contents in [Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-26.
- Updated the contents in [Compiling Embedded Module Definitions](#) on page 15-37.
- Updated the contents in [Compiling a Module Definition File](#) on page 15-42.
- Updated the contents in [Module File Errors](#) on page 15-74.
- Minor changes across [Section 15, C/C++ Program Compilation](#).
- Updated the contents in [OSS-Hosted SQL/MX COBOL Preprocessor](#) on page 16-13.
- Updated the contents in [Compiling Embedded Module Definitions](#) on page 16-25.
- Updated the contents in [Compiling a Module Definition File](#) on page 16-30.
- Updated the contents in [Module File Errors](#) on page 16-52.
- Minor changes across [Section 16, COBOL Program Compilation](#).
- Updated the contents in [Generating Locally or Globally Placed Modules](#) on page 17-3.
- Added [Generating modules in a user-specified location](#) on page 17-6.
- Updated the contents in [Specifying the search locations of the module files](#) on page 17-13.



About This Manual

This manual describes the NonStop SQL/MX programmatic interface for the ANSI C and COBOL languages. It also includes some C++ language constructs. With this interface, an application can access a database by using embedded SQL statements.

Throughout this manual, references to NonStop SQL/MX Release 2.x indicate SQL/MX Release 2.3, and subsequent releases until otherwise indicated in a replacement publication.

Audience

This manual is intended for application programmers who are embedding SQL/MX statements in a C, C++, or COBOL application. The reader should be familiar with SQL/MX terms and concepts and the American National Standards Institute (ANSI) database language SQL:1999.

- ANSI C and C++: C programmers should write to the ANSI standard for code portability. C programmers can use some, but not all, C++ language constructs in embedded SQL applications.
- ANSI COBOL85: COBOL programmers should write to the ANSI COBOL85 standard for code portability.

Organization

[Section 1, Introduction](#)

Introduces the SQL/MX programmatic interface for applications written in ANSI C/C++ or COBOL.

[Section 2, Embedded SQL Statements](#)

Describes conventions and guidelines for embedding SQL statements in an application.

[Section 3, Host Variables in C/C++ Programs](#)

Describes how to declare and use host variables in a C/C++ application.

[Section 4, Host Variables in COBOL Programs](#)

Describes how to declare and use host variables in a COBOL application.

[Section 5, Simple and Compound Statements](#)

Describes how to access data in the database by using simple DML statements.

[Section 6, Static SQL Cursors](#)

Describes how to access data in the database by using static SQL cursors.

[Section 7, Static Rowsets](#)

Describes how to use rowsets to retrieve multiple rows from the database into the application for processing and for transferring multiple rows of values from the application to the database.

[Section 8, Name Resolution, Similarity Checks, and Automatic Recompilation](#)

Describes late name resolution, similarity check, and automatic SQL recompilations.

[Section 9, Dynamic SQL](#)

Introduces dynamic SQL and describes how to write dynamic SQL applications that prepare and execute statements with dynamic input parameters.

[Section 10, Dynamic SQL With Descriptor Areas](#)

Describes how to write dynamic SQL applications by using descriptor areas.

[Section 11, Dynamic SQL Cursors](#)

Describes how to write dynamic SQL applications by using dynamic cursors.

[Section 12, Dynamic SQL Rowsets](#)

Describes how to use rowsets in a dynamic SQL environment.

[Section 13, Exception Handling and Error Conditions](#)

Describes how to get error and warning information from the SQLSTATE variable, how to use the WHENEVER exception declaration, and how to use the GET DIAGNOSTICS statement.

[Section 14, Transaction Management](#)

Describes how to use the HP NonStop Transaction Management Facility (TMF) product to ensure data consistency.

[Section 15, C/C++ Program Compilation](#)

Describes the SQL/MX compilation components and how to run the SQL/MX C/C++ preprocessor and SQL/MX compiler.

[Section 16, COBOL Program Compilation](#)

Describes how to run the SQL/MX COBOL preprocessor and SQL/MX compiler.

[Section 17, Program and Module Management](#)

Describes program and module management features and functions.

[Appendix A, C Sample Programs](#)

Describes the SQL/MX embedded SQL C sample programs.

[Appendix B, C++ Sample Program](#)

Describes the SQL/MX embedded SQL C++ sample program.

[Appendix C, COBOL Sample Programs](#)

Describes the SQL/MX embedded SQL COBOL sample programs.

Related Documentation

This manual is part of the HP NonStop SQL/MX library of manuals. The following table describes the list of manuals:

Introductory Guides

*SQL/MX Comparison Guide
for SQL/MP Users*

Describes SQL differences between NonStop SQL/MP and NonStop SQL/MX.

SQL/MX Quick Start

Describes basic techniques for using SQL in the SQL/MX conversational interface (MXCI). Includes information about installing the sample database.

Reference Manuals

SQL/MX Reference Manual

Describes the syntax of SQL/MX statements, MXCI commands, functions, and other SQL/MX language elements.

SQL/MX Messages Manual

Describes SQL/MX messages.

SQL/MX Glossary

Defines SQL/MX terminology.

Installation Guides

*SQL/MX Installation and
Upgrade Guide*

Describes how to plan for, install, create, and upgrade a SQL/MX database.

*SQL/MX Management
Manual*

Describes how to manage a SQL/MX database.

NSM/web Installation Guide

Describes how to install NSM/web and troubleshoot NSM/web installations.

Connectivity Manuals

*SQL/MX Connectivity
Service Manual*

Describes how to install and manage the HP NonStop SQL/MX Connectivity Service (MXCS), which enables applications developed for the Microsoft Open Database Connectivity (ODBC) application programming interface (API) and other connectivity APIs to use NonStop SQL/MX.

*SQL/MX Connectivity
Service Administrative
Command Reference*

Describes the SQL/MX administrative command library (MACL) available with the SQL/MX conversational interface (MXCI).

*ODBC/MX Driver for
Windows*

Describes how to install and configure HP NonStop ODBC/MX for Microsoft Windows, which enables applications developed for the ODBC API to use NonStop SQL/MX.

Migration Guides

*HP NonStop SQL/MP to
SQL/MX Database and
Application Migration Guide*

Describes how to migrate databases and applications from SQL/MP to SQL/MX.

*NonStop NS-Series
Database Migration Guide*

Describes how to migrate NonStop SQL/MX, NonStop SQL/MP, and Enscribe databases and applications to HP Integrity NonStop NS-series systems.

Data Management Guides

<i>SQL/MX Data Mining Guide</i>	Describes the SQL/MX data structures and operations to carry out the knowledge-discovery process.
<i>SQL/MX Report Writer Guide</i>	Describes how to produce formatted reports using data from a SQL/MX database.
<i>DataLoader/MX Reference Manual</i>	Describes the features and functions of the DataLoader/MX product, a tool to load SQL/MX databases.

Application Development Guides

<i>SQL/MX Programming Manual for C and COBOL</i>	Describes how to embed SQL/MX statements in ANSI C and COBOL programs.
<i>SQL/MX Query Guide</i>	Describes how to understand query execution plans and write optimal queries for a SQL/MX database.
<i>SQL/MX Queuing and Publish/Subscribe Services</i>	Describes how NonStop SQL/MX integrates transactional queuing and publish/subscribe services into its database infrastructure.
<i>SQL/MX Guide to Stored Procedures in Java</i>	Describes how to use stored procedures that are written in Java within NonStop SQL/MX.

Online Help

<i>Reference Help</i>	Overview and reference entries from the <i>SQL/MX Reference Manual</i> .
<i>Messages Help</i>	Individual messages grouped by source from the <i>SQL/MX Messages Manual</i> .
<i>Glossary Help</i>	Terms and definitions from the <i>SQL/MX Glossary</i> .
<i>NSM/web Help</i>	Context-sensitive help topics that describe how to use the NSM/web management tool.
<i>Visual Query Planner Help</i>	Context-sensitive help topics that describe how to use the Visual Query Planner graphical user interface.
<i>SQL/MX Database Manager Help</i>	Contents and reference entries from the SQL/MX Database Manager User Guide.

The NSM/web, SQL/MX Database Manager, and Visual Query Planner help systems are accessible from their respective applications. You can download the Reference, Messages, and Glossary online help from the HP Software Depot at <http://www.software.hp.com>. For more information about downloading the online help, see the *SQL/MX Release 3.2 Installation and Upgrade Guide*.

These manuals are part of the SQL/MP library of manuals and are essential references for information about SQL/MP Data Definition Language (DDL) and SQL/MP installation and management:

Related SQL/MP Manuals

<i>SQL/MP Reference Manual</i>	Describes the SQL/MP language elements, expressions, predicates, functions, and statements.
<i>SQL/MP Installation and Management Guide</i>	Describes how to plan, install, create, and manage an SQL/MP database. Describes installation and management commands and SQL/MP catalogs and files.

Examples in This Manual

The examples in this manual are written in C/C++ and COBOL.

Unless otherwise stated, all C examples use the default SQL/MX VARCHAR.

Many examples in this manual are incorporated into the complete C and C++ programs in [Appendix A, C Sample Programs](#), [Appendix B, C++ Sample Program](#), and into the complete COBOL programs in [Appendix C, COBOL Sample Programs](#).

Note. Many of the examples in this manual use the SQL/MX Release 2.x sample database, which uses SQL/MX format tables. To install the sample database, you must have a license to use SQL/MX DDL statements. To acquire the license, purchase product T0394. Without this product, you cannot install the sample database; an error message informs you that the system is not licensed.

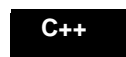
Notation Conventions

Icons

Icons that appear in the left margins of this manual represent a specific context of the SQL/MX syntax and semantics:



Designates information that is specific to embedding SQL/MX statements in C programs.



Designates information that is specific to embedding SQL/MX statements in C++ programs.



Designates information that is specific to embedding SQL/MX statements in COBOL programs.

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This data type is described under [Interval Data Type](#) on page 3-2.

General Syntax Notation

The following list summarizes the notation 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:

MAXATTACH

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

file-name

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

myfile.c

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

pathname

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

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

INT[ERRUPTS]

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

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

K [X | D] address

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned

braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

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

ALLOWSU { ON | OFF }
```

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

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

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

```
M address [ , new-value ]...
[ - ] { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }...
```

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

```
"s-char..."
```

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

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

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

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

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

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

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

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

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

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

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

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

Change Bar Notation

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

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

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

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

1 Introduction

NonStop SQL/MX is a relational database management system (RDBMS) that uses SQL:1999 to define and manipulate data in an SQL/MX database. SQL-92 is the current ANSI version of SQL (structured query language).

To access data, you execute SQL statements interactively by using the SQL/MX conversational interface (MXCI) or programmatically by embedding SQL statements in a host program written in ANSI C/C++ or COBOL.

When you embed SQL statements and declarations in a 3GL program, you can access a database by using SQL statements and then use 3GL statements to process and manipulate the data. Embedding SQL in programs enables you to build real-world applications that manipulate data within an SQL/MX database.

Note. From SQL/MX 2.3.4 onwards, NonStop SQL/MX supports embedded SQL statements in a DLL.

This section provides these overviews:

- [Referencing Database Object Names](#) on page 1-1
- [Embedding SQL Statements](#) on page 1-2
- [Declaring and Using Host Variables](#) on page 1-3
- [Using DML Statements to Manipulate Data](#) on page 1-4
- [Declaring and Using Static SQL Cursors](#) on page 1-5
- [Using DML Statements With Rowsets](#) on page 1-8
- [Processing Exception Conditions](#) on page 1-10
- [Ensuring Data Consistency](#) on page 1-11
- [Using Dynamic SQL](#) on page 1-7
- [Compiling and Building an Application](#) on page 12

Referencing Database Object Names

NonStop SQL/MX Release 2.x

In SQL/MX Release 2.x, all SQL/MX database objects use three-part ANSI names of the form *catalog.schema.name*. NonStop SQL/MX supports these database objects: base table, index, DDL lock, SQLMP alias, stored procedure, trigger, trigger temporary table, view, partition, and constraints (check constraint, not null constraint, primary key constraint, unique constraint, and referential constraint). The MPALIAS table is not needed and not used in SQL/MX Release 2.x.

NonStop SQL/MX Release 1.x

In releases prior to SQL/MX Release 2.x, to enable the use of ANSI logical names in an SQL/MP table, a user table name MPALIAS is created at installation time to store mappings from logical object names to physical Guardian locations. Use the CREATE

SQLMP ALIAS statement within your application to create the needed mappings from logical to physical names:

```
CREATE SQLMP ALIAS catalog.schema.name
                        [\node.]$volume.subvol.filename
```

When this statement is executed, a mapping is inserted as a row in the MPALIAS table. Examples of the CREATE SQLMP ALIAS statement appear in the *SQL/MX Reference Manual*.

Embedding SQL Statements

To code a 3GL application program to access data in an SQL/MX or in an SQL/MP database, use embedded SQL statements to receive data from or to insert data into a database. Use embedded SQL declarations to declare host language variables that these SQL statements use.

Note. NonStop SQL/MX does not support mixing embedded SQL calls to SQL/MX and SQL/MP from the same application process.

Your embedded SQL host program might look something like this:

```
EXEC SQL embedded SQL declarations ...
...
host language statement
...
EXEC SQL embedded SQL statement ...
...
```

In C/C++ and COBOL programs, the keywords EXEC SQL begin an embedded SQL declaration or statement.

In a C/C++ program, the semicolon (;) ends a declaration or statement. In a COBOL program, the keywords END-EXEC end a declaration or statement.

You can embed static or dynamic SQL statements in a host language source file. You code a static SQL statement as an actual SQL statement and run the SQL/MX 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, compile, and execute the SQL statement at run time.

For a list of SQL statements you can embed in a program, see [Section 2, Embedded SQL Statements](#).

Embedding SQL Statements in DLL

SQL/MX allows you to embed SQL statements in DLLs to build modular and manageable products. SQL statements can be embedded in both Guardian and OSS DLLs.

To build DLLs with embedded SQL statements, follow the compilation steps specified in [15, C/C++ Program Compilation](#) and [16, COBOL Program Compilation](#), and then

modify the `eld` options to link the application to a DLL instead of an executable. For more information, see the *DLL Programmer's Guide for TNS/E Systems*.

For efficient management of module files, SQL/MX allows the modules to be located with the corresponding DLLs.

The module files are managed in the following sequence:

1. When the application is executed, SQL/MX automatically identifies the location of all the DLLs loaded by the application.
2. SQL/MX searches for module files in the locations, in the following order:
 - a. Location of the executable program
 - b. User-specified Guardian or OSS location
 - c. Location of the DLL
 - d. System global module directory called `/usr/tandem/sqlmx/USERMODULES`

Note. When DLLs are loaded from multiple locations, the order of search is not defined.

3. SQL/MX loads the module files from all the DLL locations. If, while loading the module files, SQL/MX finds a module that matches the specified name, it stops searching the module files.

Note. You must ensure that the module file names are unique across the locations of all the DLLs.

Declaring and Using Host Variables

Host variables are host language variables declared in a host language program and used in both host language statements and embedded SQL statements. You use host variables to provide communication between 3GL and SQL statements—to receive data from a database or to insert data into a database. A host variable can be any valid host language variable that has a corresponding SQL data type.

Declaring Host Variables

Declare host variables in a Declare Section in the variable declarations part of your program. A Declare Section begins with `BEGIN DECLARE SECTION` and ends with `END DECLARE SECTION`.

Example

In this example, `hv_this_customer` and `hv_custname` are host variables:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) hv_this_customer;    /* host variables */
    char                hv_custname[19];
EXEC SQL END DECLARE SECTION;
```

In this example, HV_THIS_CUSTOMER and HV_CUSTNAME are host variables:

```
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01 HV-THIS-CUSTOMER    PIC 9(4) COMP.
        01 HV-CUSTNAME         PIC X(18).
EXEC SQL END DECLARE SECTION END-EXEC.
```

Using Host Variables

When you specify a host variable in an SQL statement, you must precede the host variable name with a colon (:). In a 3GL statement, you do not need the colon.

This example shows a host variable in an embedded SQL statement:

Example

```
C EXEC SQL SELECT custname
      INTO :hv_custname
      FROM customer
      WHERE custnum = :hv_this_customer;
...
strcpy(new_name, hv_custname);
```

The host variable hv_custname is preceded by a colon (:) in the SQL statement. In the strcpy function call, hv_custname is not preceded by a colon.

```
COBOL EXEC SQL SELECT custname
      INTO :HV-CUSTNAME
      FROM customer
      WHERE custnum = :HV-THIS-CUSTOMER
END-EXEC.
...
MOVE HV-CUSTNAME TO NEW-NAME.
```

The host variable HV-CUSTNAME is preceded by a colon (:) in the SQL statement. In the MOVE statement, HV-CUSTNAME is not preceded by a colon.

See [Section 3, Host Variables in C/C++ Programs](#), and [Section 4, Host Variables in COBOL Programs](#).

Using DML Statements to Manipulate Data

Use simple DML statements in your application program to retrieve and modify data in an SQL/MX database.

You can first test DML statements by using MXCI, the SQL/MX conversational interface. The SQL statements you enter within MXCI do not include the use of host variables, and SELECT results returned by MXCI are presented to you in the form of a result table. However, despite these differences, you can verify much of the coding of an SQL statement before embedding the statement in your program.

Examples

In these C examples, a semicolon (;) ends an embedded SQL statement. In a COBOL program, the keyword `END-EXEC` ends an embedded SQL statement.

- Single-row `SELECT` statement

C

```
EXEC SQL SELECT custname
        INTO :hv_custname
        FROM sales.customer
        WHERE custnum = :hv_this_customer;
```

The result of the `SELECT` is placed into a host variable. The selection of the single row is based on the value of the primary key (`CUSTNUM` column) as provided by the host variable.

- `INSERT` statement

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

The values of the columns inserted into the `JOB` table are provided by host variables.

- Searched `UPDATE` statement

```
EXEC SQL UPDATE persnl.employee
        SET salary = salary * :hv_inc
        WHERE salary < :hv_min_salary;
```

The `SALARY` column of all employees below a minimum salary is multiplied by a specified factor. The values of the minimum salary and the factor are provided by host variables.

- Searched `DELETE` statement

```
EXEC SQL DELETE FROM invent.partsupp
        WHERE partnum BETWEEN :hv_first_num AND :hv_last_num;
```

The rows whose part numbers are between two specified numbers are deleted from the `PARTSUPP` table. The values for the lower and upper part numbers are provided by host variables.

See [Section 5, Simple and Compound Statements](#).

Declaring and Using Static SQL Cursors

Because your 3GL program cannot handle unlimited sets of data, to retrieve data from a set of rows into your application program and then process data from that set, you must process the set one row at a time. You do this by using a cursor.

A cursor is like a pointer that traverses the set of rows in the result table of a `SELECT` statement. You specify the `SELECT` statement when you declare the cursor. You establish the result table of the `SELECT` when you open the cursor. You then fetch the

rows of the result table one at a time by using the cursor. Finally, after processing the rows, you release the result table when you close the cursor.

Examples

In these C examples, a semicolon (;) ends an embedded SQL statement. In a COBOL program, the keyword `END-EXEC` ends an embedded SQL statement.

- **DECLARE CURSOR**

C

```
EXEC SQL DECLARE get_customer CURSOR FOR
    SELECT custname, street, city, state, postcode
    FROM persnl.customer
    WHERE postcode = :hv_postcode;
```

`DECLARE CURSOR` is a preprocessor declarative, not an executable statement. It specifies that, when `OPEN` executes for this cursor, the `SELECT` statement returns five columns of data where the rows are selected by postal code. The value of postal code is provided by a host variable.

- **OPEN statement**

```
EXEC SQL OPEN get_customer;
```

The `OPEN` statement establishes the result table of `SELECT`. The selection of the rows is determined by the current value of the host variable or variables. `OPEN` positions the cursor before the first row of the result table.

- **FETCH statement**

```
EXEC SQL FETCH get_customer
    INTO :hv_custname, :hv_street, :hv_city,
        :hv_state, :hv_postcode;
```

The `FETCH` statement positions the cursor on the next row of the result table, retrieves values from that row, and places the values in the host variables. The cursor is positioned at the retrieved row.

- **Positioned DELETE statement**

```
EXEC SQL DELETE FROM persnl.customer
    WHERE CURRENT OF get_customer;
```

The `DELETE` statement deletes a single row at the current position of the cursor and positions the cursor before the next row in the result table.

- **Positioned UPDATE statement**

```
EXEC SQL UPDATE persnl.customer
    SET credit = 'A1'
    WHERE CURRENT OF get_customer;
```

The `UPDATE` statement updates values in a single row at the current position of the cursor. The cursor remains positioned on the current row.

- CLOSE statement

```
EXEC SQL CLOSE get_customer;
```

The CLOSE statement releases the result table established by OPEN for the cursor.

See also [Section 6, Static SQL Cursors](#).

Using Dynamic SQL

A static SQL statement is embedded in a host program and known at the time the host program is preprocessed. A dynamic SQL statement is either prepared dynamically with the PREPARE statement or executed through the EXECUTE IMMEDIATE statement.

Sometimes an SQL statement is not known when the program is coded—it is generated during program execution. In this case, you code a host variable with character string data type as a placeholder for an SQL statement within a PREPARE statement.

The source form of the SQL statement is a character string passed to NonStop SQL/MX for compilation with PREPARE. The character string must be a valid SQL statement. To construct the SQL statement, the program usually requires some input from a user.

A dynamic SQL program typically includes declarations and statements to:

- Declare a host variable as a place to store a dynamic SQL statement.
- Construct the SQL statement and store the statement in the declared host variable.
- Prepare the SQL statement.
- Execute the prepared statement.

See [Section 10, Dynamic SQL With Descriptor Areas](#).

A dynamic SQL program can also use either SQL descriptor areas or dynamic SQL cursors. Both techniques allow the user to specify SQL statements at run time.

Using Descriptor Areas for Dynamic SQL

A dynamic SQL program that uses descriptor areas typically includes declarations and statements to:

- Declare a host variable as a place to store a dynamic SQL statement.
- Allocate the SQL descriptor area or areas for use by dynamic parameters.
- Construct the SQL statement and store the statement in the declared host variable.
- Prepare the SQL statement.
- Describe the prepared statement using the SQL descriptor area or areas.
- Set input parameter values in the input SQL descriptor area.
- Execute the prepared statement.
- Retrieve output parameter values (if any) from the output SQL descriptor area.

- Deallocate resources held by the compiled statement and the SQL descriptor areas.

See [Section 9, Dynamic SQL](#).

Using Dynamic SQL Cursors

A dynamic SQL program that uses dynamic cursors typically includes declarations and statements to:

- Declare a host variable as a place to store the dynamic cursor specification.
- Prepare the cursor specification.
- Declare the cursor.
- Open the cursor.
- Retrieve the values at the cursor position.
- Close the cursor.

You can also use SQL descriptor areas with dynamic SQL cursors. If you do, you must describe the prepared cursor specification.

See [Section 10, Dynamic SQL With Descriptor Areas](#).

Using DML Statements With Rowsets

Use rowsets to retrieve multiple rows from the database into the application for further processing and to transfer multiple rows of values from the application to the database. Rowset arrays can be used only from embedded SQL programs. NonStop SQL/MX does not support rowsets from MXCI.

See [Section 7, Static Rowsets](#) and [Section 12, Dynamic SQL Rowsets](#) to learn how to use rowsets in C/C++ or COBOL programs.

Improving Performance by Using Rowsets

Typically, you can use a cursor specified by the SELECT statement to return the multiple rows that make up the result table of the SELECT to the application. Rows are returned one at a time. However, the cursor mechanism can produce significant overhead for an application retrieving many rows from the database.

Rowsets improve the performance of applications by manipulating multiple rows at once, instead of one at a time. Performance is improved because:

- The number of function calls between the application and NonStop SQL/MX is reduced by manipulating rows in sets. Network traffic is reduced because the data for several rows is sent in a single packet.
- When data is stored in an array, the application can bind all rows in a particular column with a single bind call and update or delete all rows by executing a single statement.

Declaring a Rowset

You declare a host variable array, along with its dimension, with the SQL Declare Section. A rowset array is declared for each column in a query. Each rowset array contains as many elements as are contained in the rowset.

Example

In this example, `hvarray_jobcode` and `hvarray_jobdesc` are host variable arrays to be used in a rowset:

C

```
EXEC SQL BEGIN DECLARE SECTION;
      ROWSET [20] unsigned NUMERIC (4)  hvarray_jobcode;
      ROWSET [20] char                    hvarray_jobdesc[19];
      ...
EXEC SQL END DECLARE SECTION;
```

Using a Rowset in a Query

You do not need to use a cursor when you are retrieving the results of a query in an output rowset and the number of rows returned does not exceed the size of the rowset.

Example

In this example, using the SQL Declare Section from the previous example, a maximum of 20 rows are retrieved from the JOB table:

C

```
EXEC SQL SELECT jobcode, jobdesc
      INTO :hvarray_jobcode, :hvarray_jobdesc
      FROM persnl.job;
```

The previous example is correct only if the SELECT INTO statement is certain to return fewer than 20 rows. If the SELECT statement can return more rows than are allocated in the rowset array, you have these choices:

- You can limit the SQL query so that it returns only a specified number of rows as shown in this example:

```
...
EXEC SQL
      SELECT [first 20]jobcode, jobdesc
      INTO :hvarray_jobcode, :hvarray_jobdesc
      FROM persnl.job;
...
```

- If you want to get all the results from the SELECT statement, use a rowset cursor. See [Selecting Rowsets With a Cursor](#) on page 7-16.

You must use a cursor when the maximum number of result rows cannot be estimated or when the memory requirements are too large to store the result table of the query.

Processing Exception Conditions

Your host language program can detect exception conditions and diagnostics information after the execution of each SQL statement. For more details, see [Section 13, Exception Handling and Error Conditions](#).

To process exception conditions and diagnostics information:

- Check the SQLSTATE variable.
- Use the WHENEVER declaration.
- Use the GET DIAGNOSTICS statement.

Checking SQLSTATE

Check the value of SQLSTATE after the execution of an SQL statement. NonStop SQL/MX returns a value to SQLSTATE to indicate the results of the execution. Your program can then use conditional statements to test the value and take appropriate action.

Example

```
C char SQLSTATE_OK[6] = "00000";
char SQLSTATE_NODATA[6] = "02000";
...
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT ... ;                /* SELECT INTO statement */
if (strcmp(SQLSTATE, SQLSTATE_NODATA) == 0) handle_nodata();
...
```

Note. Although NonStop SQL/MX supports the SQLCODE variable, use the SQLSTATE variable, which complies with the SQL:1999 standard, as the preferred status code for NonStop SQL/MX.

Using WHENEVER

Use the WHENEVER declaration to specify an action when an error, warning, or no-rows-found condition occurs. Place the WHENEVER declaration anywhere in your program. The preprocessor inserts code after every SQL statement that follows a WHENEVER declaration to check values of SQLSTATE and take appropriate action.

Example

```
COBOL ...
EXEC SQL WHENEVER NOT FOUND PERFORM ROW-NOT-FOUND-7000 END-EXEC.
...
EXEC SQL SELECT ... END-EXEC.
...
```


If the SELECT statement results in an SQLSTATE value of 02000 (no data) condition, the named error routine is executed.

Using GET DIAGNOSTICS

After the execution of an SQL statement, NonStop SQL/MX places information on exception conditions into the diagnostics area. The diagnostics area has a size limit, which is a positive integer that specifies the maximum number of conditions placed into the area during the execution of the statement. You can access the information in the diagnostics area by using the GET DIAGNOSTICS statement.

c Example

```
EXEC SQL GET DIAGNOSTICS :num = NUMBER,
                        :cmdfnc = COMMAND_FUNCTION;

...
for (i = 1; i <= num; i++) {
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
                        :hv_sqlstate = RETURNED_SQLSTATE,
                        :hv_msgtext = MESSAGE_TEXT,
                        ...;
};
```

The first GET DIAGNOSTICS statement returns the number of conditions in the condition information area and the character string that identifies which SQL statement executed. The second GET DIAGNOSTICS statement returns the value of SQLSTATE and the corresponding message text, among other condition information.

See also [Section 13, Exception Handling and Error Conditions](#).

Ensuring Data Consistency

Your application can use the TMF product to ensure the consistency of an SQL/MX database against concurrent access and system failure.

A TMF transaction—a set of database changes that must be completed as a group—is the basic recovery unit of NonStop SQL/MX. The typical order of events within a transaction is:

1. The transaction is started.
2. Database changes are made.
3. Database changes are committed.

If all changes cannot be made or you do not want to complete a transaction for some other reason, you can abort the transaction and return the database to its state before the transaction started.

To ensure that a sequence of statements either executes successfully or not at all, you can define one transaction consisting of these statements by enclosing the sequence within the BEGIN WORK and COMMIT WORK statements. You can abort a transaction with the ROLLBACK WORK statement.

Alternatively, you can commit changes automatically at the end of each SQL statement by using `SET TRANSACTION AUTOCOMMIT ON` at the beginning of your program. The default for embedded SQL is `AUTOCOMMIT OFF`.

If you exit a program without using either of these methods, any uncommitted changes are automatically rolled back.

See also [Section 14, Transaction Management](#).

Compiling and Building an Application

NonStop SQL/MX provides two methods of creating a module:

- [Processing With Embedded Module Definitions](#) on page 1-12
- [Processing With Module Definition Files](#) on page 1-12

The first method is the default method of processing programs in SQL/MX Release 2.x and later product versions. The second method is the only method of processing programs in SQL/MX Release 1.8. Although identical module definitions and identical modules are produced with either technique, HP recommends that you produce modules by using embedded module definitions.

By default, modules are created in the `/usr/tandem/sqlmx/USERMODULES` directory. Command line options with `mxCompileUserModule` and `mxcmp` and the `MXCMP_PLACES_LOCAL_MODULE` default setting provide the ability to place modules in local directories.

Processing With Embedded Module Definitions

This method, which is the default method in SQL/MX Release 2.x and later product versions, does not use module definition files (`.m` files) for SQL/MX-specific information to be SQL compiled. The preprocessor reads a 3GL source file that contains C/C++ or COBOL and SQL statements and generates one file: a single, self-contained source file that contains embedded module definitions. The annotated source file contains the source statements with the SQL statements converted to comments. To produce the module, compile the source file with the host language compiler and use the `mxCompileUserModule` command-line tool to SQL compile the embedded module definition.

For more information, see:

- [Section 15, C/C++ Program Compilation](#)
- [Section 16, COBOL Program Compilation](#)
- [Section 17, Program and Module Management](#)

Processing With Module Definition Files

This method, which is the only method you can use in SQL/MX Release 1.8, generates module definition files (`.m` files). The preprocessor reads a 3GL source file that contains

C/C++ or COBOL and SQL statements and generates two files: an annotated source file and a module definition file (*source-file.m*) that contains the SQL source statements. You compile the source file with the host language compiler, and you compile the module definition file with the SQL/MX compiler (`mxcmp`). A module definition file is not created unless you choose the appropriate preprocessor options. You must use the `-x` or `-m` preprocessor options or the `SQLMX_PREPROCESSOR_VERSION` environment variable to create a module definition file.

For more information, see:

- [Section 15, C/C++ Program Compilation](#)
- [Section 16, COBOL Program Compilation](#)
- [Section 17, Program and Module Management](#).

General Instructions

1. Use a standard programming editor and create your embedded SQL C/C++ or COBOL application.
2. Run the SQL/MX C/C++ or COBOL preprocessor to:
 - a. Parse the EXEC SQL statements and replace them with call-level interface (CLI) calls.
 - b. Create embedded module definitions (the default method in SQL/MX Release 2.x) or a module definition file (as in SQL/MX Release 1.8) describing the SQL statements.
3. Run a standard C/C++ or HP COBOL compiler and linker to create the application's executable file.
4. Run the SQL/MX compiler on the executable file to create an execution plan for the SQL statements and store the plan in a module file. Use `mxCompileUserModule` on the application executable when producing embedded module definitions or `mxcmp` on the `.m` file when producing a module definition file.

SQL/MX Host Language Preprocessor

The preprocessor opens the 3GL input source file and the 3GL output source file. By default, the preprocessor writes the modified source file and the embedded module definitions in the 3GL source file. If you choose options to create a module definition file, the preprocessor also opens the module definition file. The preprocessor reads the input source file and parses the code:

- When the preprocessor recognizes a `BEGIN DECLARE SECTION`, it parses the host variables according to the allowed 3GL declaration syntax. You can use only variables declared in an SQL Declare Section as host variables, providing communication between 3GL and SQL statements.

- When the preprocessor recognizes an EXEC SQL, it finds the corresponding terminating semicolon (;) for C/C++ programs or the terminating keywords END-EXEC for COBOL programs.
- For each embedded SQL statement, the preprocessor scans the statement to find host variable references and parses the statement to determine the required CLI calls. If the statement is valid:
 - If using the default method, the preprocessor writes the embedded module definitions in the 3GL source file.
 - If you choose to create a module definition file, the preprocessor writes the output to the module definition file and the 3GL source file.

SQL/MX Compiler

The SQL/MX compiler opens the input program executable (when using Embedded Module Definitions) or the input module definition file (when using Module Definition files) and it opens the output module file that will contain the execution plans for the SQL statements and performs the following functions:

- Expands partially qualified SQL object names using the current default settings.
- Expands view definitions.
- Performs type checking for 3GL and SQL data types.
- Checks SQL object references to verify their existence.
- Determines an optimized execution plan and access path for each DML statement.
- Generates executable code for the execution plans (if the SQL objects in the statement are present at compile time) and creates a module in the `/usr/tandem/sqlmx/USERMODULES` directory (or locally placed module directory, if specified).
- Generates a list of the SQL statements in the program file, including messages.
- Returns a completion code indicating the outcome of the compilation.

Host Language Compiler

NonStop SQL/MX supports host applications written in C/C++ and COBOL. For program preparation, see [Section 15, C/C++ Program Compilation](#), and [Section 16, COBOL Program Compilation](#).

Program and Module Management

A variety of methods and features are available for managing your programs and module files. See [Section 17, Program and Module Management](#).

2

Embedded SQL Statements

You can access an SQL/MX database by embedding SQL statements in your host language program.

This section describes:

- [Syntax for Coding SQL Statements](#) on page 2-1
- [Guidelines for Coding SQL Statements](#) on page 2-1
- [Placement of SQL Statements](#) on page 2-2
- [Embedded SQL Declarations and Statements](#) on page 2-6
- [Using CONTROL Statements](#) on page 2-12
- [Static and Dynamic CONTROL Statements](#) on page 2-13

Syntax for Coding SQL Statements

To code an embedded SQL statement in your 3GL source file, use this general syntax:

```
EXEC SQL sql-statement sql-terminator
```

sql-statement

is any SQL statement shown in [Embedded SQL Declarations and Statements](#) on page 2-6.

sql-terminator

is the terminator for the SQL statement.

- For C/C++, *sql-terminator* is a semicolon (;).
- For COBOL, *sql-terminator* is END-EXEC.

Guidelines for Coding SQL Statements

Follow the same formatting and line continuation conventions for embedded SQL statements that you use for 3GL statements.

Example

COBOL

```
EXEC SQL WHENEVER SQLERROR PERFORM 9000-SQL-ERROR END-EXEC.
```

```
EXEC SQL
  SELECT custname
  INTO :HV-CUSTNAME
  FROM customer
  WHERE custnum = :HV-THIS-CUSTOMER
END-EXEC.
```

- An SQL statement can extend over several lines.
- SQL statements cannot be nested.

- SQL statements can contain SQL comments. SQL comments begin with a double hyphen (--) and end with the end of the line.
- Embedded SQL uses the continuation character of the language in which you are programming.
- SQL statements can contain host language comments:
 - C comments have the form: /* . . . */. The comment is not restricted to one line.
 - COBOL comments have the form: * . . . The asterisk (*) is in the first column of the source code line in TANDEM free format and in the seventh column of the source code line for ANSI fixed format. The comment is restricted to one line.

Note. Many of the examples in this manual use the NonStop SQL/MX Release 2.x sample database, which uses SQL/MX format tables. To install the sample database, you must have a license to use SQL/MX DDL statements. To acquire the license, purchase product T0394. Without this product, you cannot install the sample database; an error message informs you that the system is not licensed.

Placement of SQL Statements

MODULE Directive

The MODULE directive is an embedded SQL statement that specifies the name of an embedded module for the preprocessor. Place the MODULE directive at the beginning of a 3GL program and before cursor definitions and executable SQL statements. For detailed syntax, use considerations, and examples of this statement, see the MODULE directive in the *SQL/MX Reference Manual*.

Host Variable Declarations

Code the host variables that are used in SQL statements in the SQL Declare Section. In embedded SQL programs, the SQL Declare Section is equivalent to a variable declaration. You can place an SQL Declare Section wherever it is legal to place declarations in a C, C++, or COBOL program. An embedded program can contain more than one SQL Declare Section.

You can place an SQL Declare Section relative to executable SQL statements in your embedded program. For a list of executable SQL statements, see [Table 2-4](#) on page 2-8.

To code an SQL Declare Section:

- Use BEGIN DECLARE SECTION to mark the beginning of the Declare Section.
- Code the host variables used in SQL statements within the Declare Section.

- Code INVOKE directives that generate structure descriptions of tables or views within the Declare Section.
- Use END DECLARE SECTION to mark the end of the Declare Section.

For a list of SQL statements allowed in the SQL Declare Section, see [Table 2-2](#) on page 2-6. For detailed information on each statement and the proper syntax, see the *SQL/MX Reference Manual*.

C Host Variables

In a C program, you cannot include a function declaration within an SQL Declare Section. As a result, you cannot declare the arguments of a C function as host variables. To use argument values in an embedded SQL statement, you must copy the argument values to host variables.

The SQL/MX C/C++ preprocessor, which is initiated by the `mxsqlc` command, requires the EXEC SQL BEGIN... END DECLARE SECTION block to contain only host variable declarations and SQL or host language comments. Any executable code in this block is not processed and could cause the preprocessor to return error messages.

Example

C

```
EXEC SQL BEGIN DECLARE SECTION;
...
unsigned NUMERIC (4) jobcode;          /* host variables */
char          jobdesc[19];
EXEC SQL INVOKE persnl.employee AS emp_tbl;
struct emp_tbl emp;
EXEC SQL END DECLARE SECTION;
```

C++ Host Variables Within a Class Definition

In a C++ program, you can include an SQL Declare Section within a class definition to use a data member of a class as a host variable. References to host variables declared within a class definition must be in member functions of the class. In a C++ program, you cannot include a class definition within an SQL Declare Section.

Example

C++

```
class jobsql {
// class member host variables
EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) memhv_jobcode;
    char          memhv_jobdesc[19];
EXEC SQL END DECLARE SECTION;
public:
    ...
    void putjob(){
        EXEC SQL
            INSERT INTO persnl.job
                VALUES (:memhv_jobcode, :memhv_jobdesc);
```



```

    }
}; // end of jobsql class definition

```

Nonexecutable SQL Statements

You can place a specific set of static SQL statements anywhere in an embedded C, C++, or COBOL program. However, these statements affect only the compilation of the static SQL statements that they precede. For a list of these static SQL declarations and statements, see [Table 2-3](#) on page 2-7.

Code these SQL declarations anywhere in your program but with the restrictions shown in these COBOL examples:

- DECLARE CATALOG declarations—before the SQL statements with the unqualified schema names to which the declaration applies:

```

EXEC SQL DECLARE CATALOG 'samdbcat' END-EXEC.
EXEC SQL DELETE FROM persnl.employee ... END-EXEC.

```

- DECLARE SCHEMA declarations—before the SQL statements with the unqualified object names to which the declaration applies:

```

EXEC SQL DECLARE SCHEMA 'samdbcat.persnl' END-EXEC.
EXEC SQL DELETE FROM employee ... END-EXEC.

```

- DECLARE CURSOR declarations—before the associated OPEN statement and processing statements using the cursor:

```

EXEC SQL DECLARE get_employee CURSOR FOR
    SELECT empnum, jobcode, salary
    FROM employee
    WHERE deptnum = :HV-DEPTNUM
END-EXEC.
* Move value into HV-DEPTNUM
...
EXEC SQL OPEN get_employee END-EXEC.

```

- WHENEVER declarations—before the SQL statements to which the declaration applies:

```

EXEC SQL WHENEVER NOT FOUND
    PERFORM 7000-ROW-NOT-FOUND
END-EXEC.
...
EXEC SQL SELECT ... END-EXEC.

```

Executable SQL Statements

In an embedded C, C++, or COBOL program, you must place executable SQL statements within the body of the program, such as in `main()` for C or C++ programs or within the body of other functions or procedures in the program. Code the listed types of executable SQL statements as you would executable 3GL statements:

- SQL statements that process dynamic SQL

- Diagnostics statement
- Data Definition Language (DDL) statements
- Data Manipulation Language (DML) statements
- Transaction control statements
- Object naming statements
- Data Control Language (DCL) statements
- Utilities (UPDATE STATISTICS)

For a list of executable SQL statements, see [Table 2-4](#) on page 2-8.

Executable SQL Statements in C++ Programs

In a C++ program, you can include embedded SQL statements that refer to host variables declared within a class definition only in member functions of the class within the scope of the class definition. However, you can include both of these types of embedded SQL statements within the same C++ program:

- Statements that refer to host variables declared within a class definition
- Statements that refer to host variables not declared within a class definition

Example

C++

```
// Non-class member host variables
EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) nonmemhv_jobcode;
EXEC SQL END DECLARE SECTION;

...
class jobsql {
// Class member host variables
EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) memhv_jobcode;
EXEC SQL END DECLARE SECTION;
public:
void deljob(){
    memhv_jobcode = 1234;
    EXEC SQL DELETE FROM persnl.job
        WHERE jobcode = :memhv_jobcode;
}
}; // End of jobsql class definition
...
main(){
jobsql mysql; // Instantiate a member of the class jobsql
...
// Delete job code 1234
mysql.deljob();
...
// Delete another job code 5678
nonmemhv_jobcode = 5678;
EXEC SQL DELETE FROM persnl.job
    WHERE jobcode = :nonmemhv_jobcode;
} // End of main
```

In this example, a DELETE statement is executed twice within `main()`: the first time as a member function that consists of an embedded SQL statement, and the second

time as an embedded SQL statement. The member function references host variables that are class data members.

Embedded SQL Declarations and Statements

These tables list all the SQL declarations and statements that you can embed in a 3GL program:

- [Table 2-1](#) on page 2-6 describes the MODULE directive, which you should place at the beginning of a 3GL program.
- [Table 2-2](#) on page 2-6 summarizes the embedded SQL statements that you can use only in an SQL Declare Section of a 3GL program.
- [Table 2-3](#) on page 2-7 summarizes the nonexecutable SQL statements that affect other static SQL statements embedded in a 3GL program.
- [Table 2-4](#) on page 2-8 summarizes the executable SQL statements you can embed in a 3GL program.

For detailed syntax, use considerations, and examples of the embedded SQL discussed in this manual, see the *SQL/MX Reference Manual*.

Table 2-1. MODULE Directive

Statement	Description
MODULE	Specifies module name to be used for module file.

Table 2-2. Embedded SQL Statements in SQL Declare Section

Statement	Description
BEGIN DECLARE SECTION	Designates the beginning of a Declare Section for host variable declarations.
END DECLARE SECTION	Designates the end of a Declare Section.
INVOKE*	Generates a structure description of a table or view.

* Indicates the statement is an SQL/MX extension.

Table 2-3. Nonexecutable SQL Statements

Statement	Description
Catalog and Schema Declarations	
DECLARE CATALOG*	Sets default catalog for unqualified schema names in static SQL statements within a compilation unit.
DECLARE SCHEMA*	Sets default schema for unqualified object names in static SQL statements within a compilation unit.
NAMETYPE and MPLOC Attribute Declarations	
DECLARE NAMETYPE*	Sets default NAMETYPE attribute value to ANSI or NSK for static statements within a compilation unit.
DECLARE MPLOC*	Sets default Guardian volume and subvolume for unqualified physical object names in static SQL statements within a compilation unit.
Cursor Declaration	
DECLARE CURSOR	Specifies a static cursor in a host program and associates the name of the cursor with a query expression that specifies the rows to be retrieved by using the cursor.
Exception Declaration	
WHENEVER	Generates code that checks SQL statement execution for errors and an ending no-rows-found condition and specifies an action to take.
Data Control Language (DCL) Statements	
CONTROL QUERY DEFAULT*	Overwrites the contents in memory for the current process. This statement applies to static SQL.
CONTROL QUERY SHAPE*	Forces execution plans by modifying the operator tree for a prepared statement. This statement applies to static SQL.
CONTROL TABLE*	Specifies a performance-related option for DML accesses to a table or view. The options are MDAM, PRIORITY, TABLELOCK, TIMEOUT, and RESET. This statement applies to static SQL.

* Indicates the statement is an SQL/MX extension.

Table 2-4. Executable SQL Statements (page 1 of 5)

Statement	Description
SQL Statements That Process Dynamic SQL	
ALLOCATE CURSOR	Allocates an SQL cursor.
DECLARE CURSOR	Specifies a dynamic cursor in a host program and associates the name of the cursor with a query expression that specifies the rows to be retrieved by using the cursor.
ALLOCATE DESCRIPTOR	Allocates an input or output SQL descriptor area.
DEALLOCATE DESCRIPTOR	Deallocates an SQL descriptor area.
PREPARE	Prepares (compiles) a dynamic SQL statement for subsequent execution by an EXECUTE statement.
DEALLOCATE PREPARE	Deallocates a prepared statement, returns the system resources used by the statement, and enables reuse of the statement name.
DESCRIBE	Uses an SQL descriptor area to return descriptions of output variables (usually SELECT columns) from a prepared statement.
DESCRIBE INPUT	Uses an SQL descriptor area to store information on input parameters for a prepared statement.
EXECUTE	Executes a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	Prepares (compiles) and executes a dynamic SQL statement.
GET DESCRIPTOR	Retrieves information from an SQL descriptor area.
SET DESCRIPTOR	Modifies information in an SQL descriptor area.
Diagnostics Statement	
GET DIAGNOSTICS	Returns diagnostic information on the most recently executed SQL statement.
Data Definition Language (DDL) Statements	
ALTER INDEX *	Changes the file attributes of an index.
ALTER SQLMP ALIAS *	Changes the physical name of an SQL/MP table to which an existing alias is mapped.
ALTER TABLE	Changes the definition of an table.

* Indicates the statement is an SQL/MX extension.

Table 2-4. Executable SQL Statements (page 2 of 5)

Statement	Description
Data Definition Language (DDL) Statements (continued)	
ALTER TRIGGER	Changes the definition of an trigger.
CREATE CATALOG *	Defines a catalog.
CREATE INDEX *	Creates an index based on one or more columns in a table.
CREATE PROCEDURE	Defines an existing Java method as an SPJ in NonStop SQL/MX.
CREATE SCHEMA	Defines a schema.
CREATE SQLMP ALIAS *	Defines a mapping from an ANSI name to the physical name of an SQL/MP table or view.
CREATE TABLE	Defines a persistent base table.
CREATE TRIGGER	Defines a trigger.
CREATE VIEW	Defines a viewed table.
DROP CATALOG *	Destroys an empty catalog.
DROP INDEX *	Destroys an index.
DROP PROCEDURE	Removes an SPJ definition.
DROP SCHEMA	Destroys an empty schema.
DROP SQLMP ALIAS *	Destroys the mapping of an ANSI name to the physical name of an SQL/MP table.
DROP TABLE	Destroys a table.
DROP TRIGGER	Destroys a trigger.
DROP VIEW	Destroys a view.
GRANT	Defines privileges.
GRANT EXECUTE *	Defines execute privilege on a procedure.
REGISTER CATALOG *	Registers a catalog visible on the local node to the remote node.
REVOKE	Destroys privileges.
REVOKE EXECUTE *	Destroys execute privileges on a procedure.
UNREGISTER CATALOG *	Removes an empty catalog reference from a node.

* Indicates the statement is an SQL/MX extension.

Table 2-4. Executable SQL Statements (page 3 of 5)

Statement	Description
Data Manipulation Language (DML) Statements	
CLOSE	Closes a cursor.
DELETE	Deletes rows from a table or view.
FETCH	Retrieves a row by using a cursor.
INSERT	Inserts rows into a table or view.
OPEN	Opens a cursor.
SELECT	Retrieves data from tables and views.
UPDATE	Updates values in columns of a table or view.
BEGIN...END	Designates a compound statement that groups other embedded SQL statements together.
Transaction Control Statements	
BEGIN WORK*	Starts a TMF transaction.
COMMIT [WORK]	Commits all changes made to the database during the current transaction and frees any resources.
ROLLBACK [WORK]	Backs out the current transaction and frees resources.
SET TRANSACTION	Sets attributes for the next transaction— isolation level, access mode, size of diagnostics area, and whether to commit changes automatically at the end of a statement.
Object Naming Statements	
SET CATALOG	Sets default catalog for unqualified schema names in dynamic SQL statements that are prepared after this statement is executed.
SET SCHEMA	Sets default schema for unqualified object names in dynamic SQL statements that are prepared after this statement is executed.
SET NAMETYPE*	Sets default NAMETYPE attribute value to ANSI or NSK in dynamic SQL statements that are prepared after this statement is executed.

* Indicates the statement is an SQL/MX extension.

Table 2-4. Executable SQL Statements (page 4 of 5)

Statement	Description
Object Naming Statements (continued)	
SET MPLOC*	Sets default Guardian volume and subvolume for unqualified physical object names in dynamic SQL statements that are prepared after this statement is executed. The NAMETYPE must be set to NSK for this command to work.
CONTROL QUERY DEFAULT*	Modifies the content of the SYSTEM_DEFAULTS table for the current process. This statement is executable only when you use it dynamically with PREPARE and EXECUTE or with EXECUTE IMMEDIATE. This statement affects only dynamic statements that are prepared after the execution of this statement.
CONTROL QUERY SHAPE*	Forces execution plans by modifying the operator tree for a prepared statement. This statement is executable only when you use it dynamically with PREPARE and EXECUTE or with EXECUTE IMMEDIATE. This statement affects only dynamic statements that are prepared after the execution of this statement.
CONTROL TABLE*	Specifies a performance-related option for DML accesses to a table or view. The options are MDAM, PRIORITY, TABLELOCK, TIMEOUT, and RESET. This statement is executable only when you use it dynamically with PREPARE and EXECUTE or with EXECUTE IMMEDIATE. This statement affects only dynamic statements are that prepared after the execution of this statement.
SET TABLE TIMEOUT*	Sets a dynamic value for a lock timeout or a stream timeout in the environment of the current session.
LOCK TABLE*	Locks a table or underlying tables of a view and associated indexes.
UNLOCK TABLE *	Releases locks held on nonaudited tables and views.
* Indicates the statement is an SQL/MX extension.	

Table 2-4. Executable SQL Statements (page 5 of 5)

Statement	Description
Utilities	
UPDATE STATISTICS	Updates information on the content of a table and its indexes.
Stored Procedures (SPJ)	
CALL Statement	The CALL statement invokes a stored procedure in Java (SPJ) and can be embedded in a C, C++, or COBOL program. Both static and dynamic CALL statements are supported. See <i>SQL/MX Guide to Stored Procedures in Java</i> or the <i>SQL/MX Reference Manual</i> for examples and more information.
* Indicates the statement is an SQL/MX extension.	

Considerations for Embedding DDL and DML Statements

The most practical way to create your database objects is through OBEY command files in MXCI. By using OBEY scripts, you avoid compilation. However, you might have a reason for embedding DDL and DML statements within the same program. If you should do this, understand that the DDL statements create the objects at run time, not compile time. At compile time, subsequent DML statement on those objects are not compiled statically because the objects do not exist in the database at compile time. At run time, the DML statements execute dynamically.

For information on DDL and DML statements, see the *SQL/MX Reference Manual*.

Considerations for Embedding the UPDATE STATISTICS Statement

It is not recommended that your application start a transaction prior to executing UPDATE STATISTICS because UPDATE STATISTICS will execute under the user transaction. However, you can embed UPDATE STATISTICS in C programs because the C preprocessor does not start transactions automatically. Transactions are started when you use the BEGIN WORK statement in the application.

Using CONTROL Statements

CONTROL statements are SQL/MX compiler directives that affect the execution of SQL statements in a program and that enable you to override the system-level default settings for the current process. CONTROL statements include:

- CONTROL QUERY DEFAULT, which overrides system-level default settings

- CONTROL QUERY SHAPE, which forces execution plans by modifying the operator tree for a prepared statement
- CONTROL TABLE, which specifies a performance-related option for DML accesses to a table or view

For the syntax of CONTROL statements, see the *SQL/MX Reference Manual*.

ANSI Compliance and Portability

If program portability is important, note that CONTROL statements are SQL/MX extensions to the ANSI standard.

Static and Dynamic CONTROL Statements

CONTROL statement directives influence static SQL statements and dynamic SQL statements differently. As a programmer, you need to be aware of these differences to minimize confusion regarding the scope and influence of CONTROL statements in your program.

CONTROL, Line Order Scope, and Static SQL programs

In an embedded static SQL program, CONTROL statements are preceded by the preprocessor EXEC SQL directive. In this type of program, a CONTROL statement directive applies only to static SQL statements. Moreover, CONTROL statements apply to all statements that follow the CONTROL statement until superseded by another EXEC SQL CONTROL statement. This type of scope is commonly referred to as *line order scope* and applies exclusively to static SQL statements.

Note. Scoping rules behave differently for NonStop SQL/MP than for NonStop SQL/MX. Static statements are in pure listing order in NonStop SQL/MX. The scope is reset at each procedure boundary in NonStop SQL/MP. For more information about the differences between NonStop SQL/MP and NonStop SQL/MX, see the *SQL/MX Comparison Guide for SQL/MP Users*.

CONTROL, Flow Control Scope, and Dynamic SQL programs

In a dynamic SQL program, CONTROL statements are executed dynamically. In MXCI, for example, you enter CONTROL statements in the same manner as other SQL statements. You can execute CONTROL statements in a C or COBOL program with either PREPARE/EXECUTE or EXECUTE IMMEDIATE. In this type of a program, the scope is such that dynamically executed CONTROL statements apply only to dynamic SQL statements that are executed after the CONTROL statement has been executed. This type of scope is called flow control scope and applies exclusively to dynamic SQL statements. [Example 2-1](#) on page 2-14 illustrates the different types of scope and

influence that CONTROL statement directives have on dynamic and static SQL statements.

Note. As a rule, dynamic CONTROL statements apply to dynamic SQL statements that are prepared after the CONTROL statement is executed. If your program uses a PREPARE once and EXECUTE many strategy, the program must execute the CONTROL statement prior to the PREPARE statement. CONTROL statements that are executed after a PREPARE statement do not apply to the prepared statements.

Guidelines on the Behavior and Use of CONTROL Statements

- Place CONTROL statements anywhere an executable statement is allowed in a C, C++, or COBOL program.
- A subsequent CONTROL statement can modify the values set by a prior CONTROL statement.
- The CONTROL statements that are within scope stay in effect for the entire duration of the SQL statements they influence. For example, a CONTROL TABLE statement stays in effect until the current process terminates or until the execution of another CONTROL statement overrides it.
- It is good practice to issue a CONTROL QUERY SHAPE OFF statement immediately after the execution of a query with a forced plan because the next query might not fit the forced plan and result in an optimizer error. For more information, see the *SQL/MX Reference Manual*.

Example 2-1. Static and Dynamic SQL and CONTROL Scope

C

```
...
EXEC SQL CONTROL QUERY DEFAULT TABLELOCK 'ON';
EXEC SQL UPDATE sales SET SALARY = SALARY * 0.05;
...

for (i = 1; i <= 3; i++)
{
    printf("Enter SQL statement: " );
    scanf("%s", &sql_string);
    EXEC SQL PREPARE stmt FROM :sql_string;
    EXEC SQL EXECUTE stmt;
}
```

Prior to entering the loop, the static statement EXEC SQL UPDATE is affected by the preceding static CONTROL statement.

Now, consider dynamic SQL statements. The first time through the loop, the user enters:

```
UPDATE sales SET SALARY = SALARY * 0.05;
```

The preceding static CONTROL QUERY statement does not apply to this dynamic statement when the user enters this dynamic SQL statement, and the default TABLELOCK strategy (SYSTEM) is used.

The second time through the loop, the user enters:

```
CONTROL QUERY DEFAULT TABLELOCK 'ON' ;
```

When the user enters this dynamic CONTROL statement, it is executed dynamically. As discussed previously, all subsequent SQL statements are affected by this CONTROL statement.

The third and final time through the loop, the user enters:

```
UPDATE sales SET SALARY = SALARY * 0.05;
```

In this dynamic SQL statement, the preceding CONTROL QUERY statement affects the TABLELOCK strategy. This example demonstrates the behavior of flow control scope.

Host Variables in C/C++ Programs

Host variables are data items declared in a host application program and used in both host language statements and embedded SQL statements. They provide communication between SQL statements and the host language statements. An input host variable transfers data from a host language program to an SQL/MX database, and an output host variable transfers data from a database to the program.

This section describes:

- [Specifying a Declare Section](#) on page 3-1
- [C Host Variable Data Types](#) on page 3-2
- [Using Corresponding SQL and C Data Types](#) on page 3-8
- [Specifying Host Variables in SQL Statements](#) on page 3-15
- [Using Host Variables in a C/C++ Program](#) on page 3-16
- [Using Indicator Variables in a C/C++ Program](#) on page 3-41
- [Creating C Host Variables Using INVOKE](#) on page 3-43
- [Character Set Examples](#) on page 3-52

Specifying a Declare Section

Declare all host variables within an SQL Declare Section:

- Use the BEGIN DECLARE SECTION statement to begin a Declare Section.
- Use the END DECLARE SECTION statement to end a Declare Section.
- You can specify more than one Declare Section in your source file but do not nest them.
- You can use SQL or host language comments in a Declare Section.
- For the best performance, declare the host variable as the same data type as the column in the SELECT list. If you declare this way, you can use bulk moves to input and output data.
- You can declare all types of host variables as structure fields or class data members. Additionally, you can declare pointers to structures, classes, and binary numeric host variables as host variables.

Do not:

- Place a Declare Section within a C structure declaration.
- Include a C++ class definition within a Declare Section.
- Include a C function declaration within a Declare Section.
- Include any executable code within a Declare Section.

Example

This example uses host variable declarations in an SQL Declare Section:

```
C EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
unsigned NUMERIC (6) ordernum;           /* simple variables */
struct employee {                        /* structure */
    unsigned short    empnum;
    char              first_name[16];
    char              last_name[21];
    unsigned short    deptnum;
    unsigned short    jobcode;
} employee_info;
...
int * hvarPointer;                       /* pointer to numeric data type */
struct employee * empStruct;             /* structure pointer */
EXEC SQL END DECLARE SECTION;
```

C Host Variable Data Types

You must explicitly declare all host variables used in SQL statements. A host variable used in an SQL statement must be declared in an SQL Declare Section prior to the first use of the host variable in an SQL statement. Only variables of the types recognized by the 3GL preprocessor can appear within an SQL Declare Section.

To declare a C host variable, specify one of these data types:

```
char [char-set] C-identifier [l + 1]
NCHAR C-identifier [l + 1]
VARCHAR [char-set] C-identifier [l + 1]
NCHAR VARYING C-identifier [l + 1]
{[signed] | unsigned} NUMERIC (p, s)
PIC[TURE] [S]9(l-s)V9(s) COMP
{[signed] | unsigned} DECIMAL (l, s)
PIC[TURE] [S]9(l-s)V9(s)
{[signed] | unsigned} short
{[signed] | unsigned} int
{[signed] | unsigned} long
long long
float
double
DATE
TIME [(n)]
TIMESTAMP [(n)]
INTERVAL [period1[(n)] | SECOND [(n[,m])]] [TO
period2[(m)]]
```

Character Host Variables

```
char [char-set] C-identifier [l + 1] [C-initial-value] [, ...]
```

specifies the data type of a target host variable for a column of this SQL data type:

```
CHAR[ACTER] [(l) ]
```

char-set is specified as CHARACTER SET [IS] *character-set-name*. This optional clause specifies the character set to be associated with the host variable. If no *char-set* is specified, the default character set for *char* is ISO88591.

char-set can be ISO88591, UCS2, KANJI, KSC5601. Note that you can use host variables with the KANJI or KSC5601 character set in an SQL/MX application only to access KANJI or KSC5601 columns in an SQL/MP table.

C-identifier specifies that the variable can hold a fixed-length character (or *code_units*) string. The maximum length of the string is specified in the *length* field.

The length *l* + 1 is required and must be enclosed in square brackets. The length *l* corresponds to the length of the column value. To allow for the null terminator, add 1 to the length for the declaration of the host variable.

C-initial-value is a valid string literal in C/C++. The initial value should be of the same type as the C array type translated by the SQL/MX preprocessor.

```
NCHAR C-identifier [l +1] [C-initial-value] [, ...]
```

specifies the data type of a target host variable for a column with data in the pre-defined national character set of this SQL data type:

```
NCHAR(l)
```

C-identifier specifies that the variable can hold a fixed-length character (or *code_units*) string. The maximum length of the string is specified in the *length* field.

The length *l* + 1 is required and must be enclosed in square brackets. The length *l* corresponds to the length of the column value. To allow for the null terminator, add one (1) to the length for the declaration of the host variable.

NCHAR is always associated with the system default NATIONAL_CHARSET. For information on setting NATIONAL_CHARSET, see the *SQL/MX Reference Manual*.

C-initial-value is a valid string literal in C/C++. The initial value should be of the same type as the C array type translated by the SQL/MX preprocessor.

```
VARCHAR [char-set] C-identifier [l + 1] [C-initial-value] [, ...]
```

specifies the data type of a target host variable for a column of this SQL data type:

```
VARCHAR(l)
```

char-set is specified as CHARACTER SET [IS] *character-set-name*. This optional clause specifies the character set to be associated with the host variable. If no *char-set* is specified, the default character set for VARCHAR is ISO88591. *char-set* can be ISO88591, UCS2, KANJI, KSC5601. Note that host variables with the KANJI or KSC5601 character set in an SQL/MX application can be used only to access KANJI or KSC5601 columns in an SQL/MP table.

C-identifier specifies that the variable can hold a variable-length character (or code_units) string. The maximum length of the string is specified in the *length* field.

The length *l* + 1 is required and must be enclosed in square brackets. The length *l* corresponds to the maximum length of the column value. To allow for the null terminator, add 1 (one) to the length for the declaration of the host variable.

C-initial-value is a valid string literal in C/C++. The initial value should be of the same type as the C array type translated by the SQL/MX preprocessor.

```
NCHAR VARYING  C-identifier [l + 1] [C-initial-value] [, ...]
```

specifies the data type of a target host variable for a column of this SQL data type:

```
NCHAR VARYING (l)
```

C-identifier specifies that the variable can hold a variable-length character (or code_units) string. The maximum length of the string is specified in the *length* field.

The length *l* + 1 is required and must be enclosed in square brackets. The length *l* corresponds to the maximum length of the column value. To allow for the null terminator, add one (1) to the length for the declaration of the host variable.

NCHAR VARYING is always associated with the system default NATIONAL_CHARSET. For information on setting NATIONAL_CHARSET, see the *SQL/MX Reference Manual*.

C-initial-value is a valid string literal in C/C++. The initial value should be of the same type as the C array type translated by the SQL/MX preprocessor.

Date-Time and Interval Host Variables

In SQL/MX Release 2.x, you can declare ANSI-99 date-time and interval data types as host variables. You can use these host variables to retrieve date-time or interval data directly from date-time or interval columns or to put date-time or interval data directly into date-time or interval columns.

For SQL/MP DATETIME data types that are not equivalent to DATE, TIME, or TIMESTAMP, you are still required to declare a character array host variable and use the CAST function for input to and output from date-time or interval columns, similar to SQL/MX Release 1.8, which does not support ANSI-99 date-time host variables.

DATE

specifies the data type of a target host variable for a date-time column that contains a date in the external form *yyyy-mm-dd*.

TIME [(*n*)]

specifies the data type of a target host variable for a date-time column that, without the optional *n* precision, contains a time in the external form hh:mm:ss. The *n* precision is a positive integer that specifies the number of digits in the fractional seconds. The default for the precision is 0, and the maximum is 6.

TIMESTAMP [(*n*)]

specifies the data type of target host variable for a date-time column that, without the optional *n* precision, contains a timestamp in the external form:

yyyy-mm-dd hh:mm:ss

The *n* precision is a positive integer that specifies the number of digits in the fractional seconds, as shown in bold text:

yyyy-mm-dd hh:mm:ss.msssss

The default for precision is 6, and the maximum is 6.

INTERVAL [*period1*[(*n*)] | SECOND [(*n*[,*m*)]]] [TO *period2*[(*m*)]]

where

n leading precision

m fractional precision

period YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

period1 must be greater or equal time part than *period2*. YEAR to SECOND is valid. SECOND to YEAR is invalid.

Specifies a column that represents a duration of time as either a year-month or day-time range or a single-field. *period1* can have a leading-precision up to 18 digits (the maximum depends on the number of fields in the interval). The leading-precision is the number of digits allowed in *period1*. If *period2* is SECOND, it can have a fractional-precision up to 6 digits. The fractional-precision is the number of digits of precision after the decimal point. The default for leading-precision is 2, and the default for fractional-precision is 6. If the single-field is SECOND, the leading-precision is the number of digits of precision before the decimal point, and the fractional-precision is the number of digits of precision after the decimal point.

Numeric Host Variables

```
{[signed] | unsigned} NUMERIC (p, s)
```

specifies the data type of a target host variable for a column of one of these SQL data types:

```
NUMERIC [(p, s)] [SIGNED|UNSIGNED]
PIC[TURE] [S] {9(l-s) [V[9(s)] | V9(s)] COMP
```

The precision *p* corresponds to the precision of the column value. The scale *s* corresponds to the scale of the column value. The precision *p* for the NUMERIC data type cannot exceed 128. If you specify a precision greater than 128, the error "Error 13061 - EXC_BAD_UNN_NUM_PREC," is returned.

The length *l* for the PICTURE data type corresponds to the number of digits in the column value and cannot exceed 18. The value *l-s* is the number of digits in the integral part of the column value.

```
PIC[TURE] [S]9(l-s)V9(s) COMP
```

is the same as {[signed] | unsigned} NUMERIC (*p*, *s*).

```
{[signed] | unsigned} DECIMAL(l, s)
```

specifies the data type of a target host variable for a column of one of these SQL data types:

```
DECIMAL (l, s) [SIGNED|UNSIGNED]
PIC[TURE] [S] 9(l-s) V9(s) DISPLAY [SIGN IS LEADING]
```

The length *l* corresponds to the number of digits in the column value. The scale *s* corresponds to the scale of the column value. The value *l-s* is the number of digits in the integral part of the column value.

```
PIC[TURE] [S]9(l-s)V9(s)
```

is the same as {[signed] | unsigned} DECIMAL(*l*, *s*).

```
{[signed] | unsigned} short
```

specifies the data type of a target host variable for a column of the SQL data type:

```
SMALLINT [SIGNED|UNSIGNED]
```

```
{[signed] | unsigned} int
```

specifies the data type of a target host variable for a column of the SQL data type:

```
INTEGER [SIGNED|UNSIGNED]
```

```
{[signed] | unsigned} long
```

specifies the data type of a target host variable for a column of the SQL data type:

```
INT[EGER] [SIGNED|UNSIGNED} for 32-bit data model
```

```
LARGEINT for 64 bit data model
```

Note. The unsigned long data type is not supported in 64-bit embedded SQL/MX programs.

Note. The long data type corresponds to a 32-bit integer in programs compiled for the 32-bit address model and to a 64-bit integer in programs compiled for the 64-bit address model. Therefore, HP recommends that you use the int data type for 32-bit integer host variable.

```
long long
```

specifies the data type of a target host variable for a column of the SQL data type:

```
LARGEINT
```

Note. The unsigned long long data type is not supported in embedded SQL/MX programs.

Floating-Point Host Variables

With NonStop SQL/MX Release 2.x, you can choose to declare floating-point host variables with a Tandem floating-point format or the ANSI IEEE floating-point format. The storage and precision of IEEE floating-point data types is different from that of Tandem floating-point data types, as noted in these summaries. Tandem floating-point data types are stored in 4 bytes (REAL) or 8 bytes (DOUBLE), depending on their precision. ANSI IEEE floating-point data type FLOAT(p) declarations are stored in 8 bytes regardless of the precision.

Note. The floating-point format for SQL/MP tables is Tandem. The floating-point format for SQL/MX format tables is IEEE. Use the `-o` preprocessor option to change the format of data that is input or output to host variables in an embedded program. See [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

```
float
```

specifies the data type of a target host variable for a column of this SQL data type:

```
REAL
```

Tandem floating-point REAL data type is stored with 22 bits of precision and 9 bits of exponent. The precision corresponds to the precision of the column value. IEEE floating-point format REAL is stored in 4 bytes with 23 bits of binary precision and 8 bits of exponent.

`double`

specifies the data type of a target host variable for a column of one of these SQL data types:

```
FLOAT [(p)]  
DOUBLE PRECISION
```

IEEE floating-point format `FLOAT (p)` and `DOUBLE` are stored in 8 bytes with 52 bits of binary precision and 11 bits of exponent. The storage size for the IEEE floating-point format is implementation-defined. This is ANSI-compliant behavior.

The maximum precision for an IEEE data type is 52. The precision corresponds to the precision of the column value.

For the corresponding SQL and C host variable data types for `FLOAT` data types, see [Table 3-2](#) on page 3-10. For the generated C data types for `NUMERIC`, `PIC`, and `DECIMAL`, see [Table 3-1](#) on page 3-9.

Using Corresponding SQL and C Data Types

[Table 3-1](#) lists the corresponding SQL data types, C host variable data types, and translated C declarations for the `NUMERIC`, `DECIMAL`, `PIC`, `SMALLINT` and `LARGEINT` data types.

[Table 3-2](#) on page 3-10 lists the corresponding SQL data types, C host variable data types, and translated C declarations for the `FLOAT` data types.

[Table 3-3](#) on page 3-11 lists the corresponding SQL data types, C host variable data types, and translated C declarations for the date-time data types.

Most SQL data types with scale have extended host variable data types. You can specify a C/C++ variable as a host variable if it has a corresponding SQL data type.

Table 3-1. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, and LARGEINT Data Types

SQL Data Type	C Host Variable Data Type	Translated C Declaration (32-bit Address Model)	Translated C Declaration (64-bit Address Model)
NUMERIC (1 to 4, <i>s</i>) SIGNED	NUMERIC (1 to 4, <i>s</i>)*	short	short
NUMERIC (1 to 4, <i>s</i>) UNSIGNED	unsigned NUMERIC (1 to 4, <i>s</i>)*	unsigned short	unsigned short
NUMERIC (5 to 9, <i>s</i>) SIGNED	NUMERIC (5 to 9, <i>s</i>)*	long	int
NUMERIC (5 to 9, <i>s</i>) UNSIGNED	unsigned NUMERIC (5 to 9, <i>s</i>)*	unsigned long	unsigned int
NUMERIC (10 to 18, <i>s</i>) SIGNED	NUMERIC (10 to 18, <i>s</i>)*	long long	long long
NUMERIC (10 to 128, <i>s</i>) UNSIGNED	unsigned NUMERIC (10 to 128, <i>s</i>)*	Int16 [<i>x</i>]**	Int16 [<i>x</i>]**
NUMERIC (19 to 128, <i>s</i>) SIGNED	NUMERIC (19 to 128, <i>s</i>)*	Int16 [<i>x</i>]**	Int16 [<i>x</i>]**
PIC[TURE] [S] 9(1- <i>s</i>)V9(<i>s</i>) COMP	Same as NUMERIC*	Same as NUMERIC	Same as NUMERIC
DEC[IMAL] (1, <i>s</i>) SIGNED	DECIMAL (1, <i>s</i>)*	char [1 + 2]	char [1 + 2]
DEC[IMAL] (1, <i>s</i>) UNSIGNED	unsigned DECIMAL (1, <i>s</i>)*	char [1 + 2]	char [1 + 2]
PIC[TURE] [S] 9(1- <i>s</i>)V9(<i>s</i>)	Same as DECIMAL*	char [1 + 2]	char [1 + 2]
SMALLINT SIGNED	short	short	short
SMALLINT UNSIGNED	unsigned short	unsigned short	unsigned short
INT[EGER] SIGNED	int	long	int

* These host variable data types are extensions.

** *x* is computed based on precision *p*.

l A positive integer that represents the length.

s A positive integer that represents the scale of the number.

Table 3-1. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, and LARGEINT Data Types

SQL Data Type	C Host Variable Data Type	Translated C Declaration (32-bit Address Model)	Translated C Declaration (64-bit Address Model)
INT[EGER] UNSIGNED	unsigned int	unsigned long	unsigned int
NUMERIC(5-9)	long	long (32-bit integer)	long (64-bit integer)
LARGEINT	long long	long long (64-bit integer)	long long (64-bit integer)

* These host variable data types are extensions.
 ** x is computed based on precision p .
 l A positive integer that represents the length.
 s A positive integer that represents the scale of the number.

Note.

- You can declare an Unsigned Numeric variable and then store a negative BigNum value. Unsigned Numeric and Signed Numeric binary formats are the same for BigNum values, with the exception of the sign bit.
- The unsigned long data type is not supported in 64-bit embedded SQL/MX programs.

Table 3-2. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for Float Data Types

SQL Data Type	C Host Variable Data Type	Translated C Declaration
REAL FLOAT (1 to 22 bits)	float*	float (Tandem format)
FLOAT (23 to 54 bits) DOUBLE PRECISION	double*	double (Tandem format)
REAL	float**	float (IEEE format)
FLOAT (1 to 52 bits) DOUBLE PRECISION	double**	double (IEEE format)

* Tandem floating point is specified during compilation.
 ** IEEE floating point is specified during compilation.

Table 3-3. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations for Date-Time Data Types

SQL Data Type	C Host Variable Data Type	Translated C Declaration
DATE	DATE*	char [l + 1]
TIME (<i>time-precision</i>)	TIME[(n)]*	char [l + 1]
TIMESTAMP (<i>timestamp-precision</i>)	TIMESTAMP[(n)]*	char [l + 1]
INTERVAL { <i>start-field</i> TO <i>end-field</i> <i>single-field</i> }	INTERVAL [<i>period1</i> [(n)] SECOND [(n[,m])]] [TO <i>period2</i> [(m)]]**	char [l + 1]

* An extra character is generated for a null terminator. For DATE, the value of the length *l* is 10. For TIME(6), the value of the length *l* is 15. For TIMESTAMP(6), the value of the length *l* is 26.

** The INTERVAL data type has an extra character for a null terminator. The sign is included in the length *l*.

Extended Host Variable Data Types and Generated C Data Types

[Table 3-4](#) on page 3-12 lists the embedded SQL/C host variable "a[100]" with all its legal SQL/MX modifiers, the equivalent data types in NonStop SQL/MX, and the translated C declarations. The assumed wchar_t size is 2 characters.

Table 3-4. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations (page 1 of 2)

SQL Data Type	C Host Variable Data Type	Translated C Declaration
CHAR(99) CHARACTER SET ISO88591	char a[100]* char CHARACTER SET IS ISO88591 a[100] char CHARACTER SET IS ISO88591 a[100 CHARACTERS]	char a[100]
CHAR(99) CHARACTER SET UCS2	char CHARACTER SET IS UCS2 a[100] char CHARACTER SET IS UCS2 a[100 CHARACTERS]	wchar_t a[100]
CHAR(99) CHARACTER SET KANJI**	char CHARACTER SET IS KANJI a[100] char CHARACTER SET IS KANJI a[100 CHARACTERS]	wchar_t a[100]
CHAR(99) CHARACTER SET KSC5601**	char CHARACTER SET IS KSC5601 a[100] char CHARACTER SET IS KSC5601 a[100 CHARACTERS]	wchar_t a[100]
VARCHAR(99) CHARACTER SET ISO88591	VARCHAR a[100]* VARCHAR CHARACTER SET IS ISO88591 a[100] VARCHAR CHARACTER SET IS ISO88591 a[100 CHARACTERS]	char a[100]
VARCHAR(99) CHARACTER SET UCS2	VARCHAR CHARACTER SET IS UCS2 a[100] VARCHAR CHARACTER SET IS UCS2 a[100 CHARACTERS]	wchar_t a[100]
* An extra character is generated as a placeholder for a null terminator. The embedded SQL C VARCHAR data type is SQL:1999.		
** KANJI and KSC5601 character sets can be used only with SQL/MP tables.		

Table 3-4. Corresponding SQL, C Host Variable Data Types, and Translated C Declarations (page 2 of 2)

SQL Data Type	C Host Variable Data Type	Translated C Declaration
VARCHAR(99) CHARACTER SET KANJI**	VARCHAR CHARACTER SET IS KANJI a[100] VARCHAR CHARACTER SET IS KANJI a[100 CHARACTERS]	wchar_t a[100]
VARCHAR(99) CHARACTER SET KSC5601**	VARCHAR CHARACTER SET IS KSC5601 a[100] VARCHAR CHARACTER SET IS KSC5601 a[100 CHARACTERS]	wchar_t a[100]
CHAR(99) CHARACTER SET X	NCHAR a[100 CHARACTERS]	Depends on X (X is the value of the system default NATIONAL_CHARSET)
VARCHAR(99) CHARACTER SET X	NCHAR VARYING a[100 CHARACTERS]	Depends on X (X is the value of the system default NATIONAL_CHARSET)
* An extra character is generated as a placeholder for a null terminator. The embedded SQL C VARCHAR data type is SQL:1999.		
** KANJI and KSC5601 character sets can be used only with SQL/MP tables.		

Data Conversion

NonStop SQL/MX 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/MX 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/MX converts the value to the data type of the host variable.

NonStop SQL/MX supports conversion within numeric types and character types—but not between numeric and character types.

Converting Numeric Types

Values of data types NUMERIC, DECIMAL, PICTURE 9's, INTEGER, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION are numbers and are all mutually comparable and mutually assignable.

NonStop SQL/MX converts data between signed and unsigned numeric types and between numeric types with different precisions. Note that if a signed numeric type has a negative value, it cannot be converted.

If assignment would result in a loss of significant digits, NonStop SQL/MX returns a data exception condition in SQLSTATE. For a description of SQLSTATE values, see [Table 13-1](#) on page 13-2.

Converting Character Types

Values of data types CHARACTER, PICTURE X's, and CHARACTER VARYING are character strings and are all mutually comparable and mutually assignable if both are of the same character set. In addition, UCS2 host variables are mutually comparable and assignable with ISO88591 nonhost variable objects.

For character strings of different lengths, NonStop SQL/MX pads the receiving string variable on the right with blanks as necessary.

If the receiving string variable is too short, NonStop SQL/MX truncates the right part of the string retrieved from the database and returns a data exception condition in SQLSTATE. For a description of SQLSTATE values, see [Table 13-1](#) on page 13-2.

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 character required for the null terminator). This programming practice minimizes the data conversion performed by NonStop SQL/MX and therefore can improve the performance of your program.

Host Variable Pointers

The following types of pointers can be declared as host variables:

- Pointer to a structure.
- Pointer to a binary fixed-point numeric host variable.
- Pointer to a floating-point numeric host variable.

Do not declare the following data type host variables as pointers:

- Character
- Decimal numeric
- Date-time or interval data

If you need to access a host variable with any of these data types through a pointer, you must make the pointer a structure field.

Example

The following example shows that you cannot declare a character data type as a host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
/* cannot declare a character type pointer as host variable */
char * charPtr;
...
```

This is an illegal construct and returns the following warnings when compiled:

Hewlett-Packard NonStop(TM) SQL/MX C/C++ Preprocessor 3.2.1

(c) Copyright 2003, 2004-2013 Hewlett-Packard Development Company, LP.

*** WARNING[13029] Pointer or Reference types not supported for Host Variables.

*** WARNING[13025] Warning(s) near line 5.

The following example shows the use of character data type in a structure:

```
EXEC SQL BEGIN DECLARE SECTION;
struct ptrType
{
/* all data types are allowed as structure fields */
  char charField[100];
  ...
};
struct ptrType * structPtr;
...
```

Specifying Host Variables in SQL Statements

Use host language naming conventions for your host variable and indicator variable names. For example, a name in a C program contains alphanumeric characters, including the underscore (_), and begins with a letter or an underscore. To avoid conflicts with system-generated names, do not begin your host variable names with underscores.

After you declare a host variable, to specify it within an embedded SQL statement, use this syntax:

```
:variable-name [[INDICATOR] :indicator_variable]
```

variable-name

is the host variable name. It can be any valid host language identifier with a data type that corresponds to an SQL data type. You must precede *variable-name* with a colon (:) within an SQL statement.

INDICATOR

is a keyword that can precede *indicator_variable*.

indicator_variable

is an indicator variable of exact numeric data type. You must declare the indicator variable as type `short` in C. You must precede *indicator_variable* with a colon (:) in an SQL statement.

If data returned in the host variable is null, the indicator variable is less than zero. If character data returned is truncated, the indicator variable is set to the length of the string in the database. Otherwise, the value of the indicator variable is zero. To insert null into the database, set the indicator variable to a value less than zero.

Using Host Variables in a C/C++ Program

As a C/C++ programmer, you need to know how to declare and use host variables to retrieve and insert data with these SQL data types:

- [Fixed-Length Character Data](#) on page 3-17
- [Variable-Length Character Data](#) on page 3-19
- [Numeric Data](#) on page 3-23
- [Date-Time and Interval Data](#) on page 3-34

In a C program, you can use structures for host variables. When you refer to a single field name in the structure, you must include the structure name with the field name.

In a C++ program, you can use a data member of a class as a host variable. References to host variables declared within a class definition must be within member functions of the class. See [Host Variables as Data Members of a C++ Class](#) on page 3-40.

Character Set Data

These guidelines apply for NonStop SQL/MX Release 1.8 and NonStop SQL/MX Release 2.x character sets:

- ISO88591 character set: An SQL/MX Release 1.8 application can be run under SQL/MX Release 2.x without application recompilation, if the application contains ISO88591 character data only.
- KANJI and KSC5601 character set: If KANJI or KSC5601 character set host variables are contained in the application, the application must be carefully rewritten and recompiled. KANJI and KSC5601 host variables in C applications are translated as single-byte arrays in SQL/MX Release 1.8 and as double-byte arrays in SQL/MX Release 2.x. If the application is not rewritten, SQL errors might be emitted, corruption of data might occur, and the application might crash.

Host variable source code in SQL/MX Release 1.8:

```
char a[100]; /* Host variable stores KANJI character strings */
```

Host variable source code in SQL/MX Release 2.x:

```
char CHARACTER SET IS KANJI a[100];
```

Guidelines for Revising KANJI/KSC5601 Character Set Host Variables

- Use the character set clause `CHARACTER SET IS KANJI` or `CHARACTER SET IS KSC5601`.
- The encoding for KANJI is the double-byte subset of the Shift-JIS, with no check on code points performed by NonStop SQL/MX. For the best results, use the big-endian byte order to denote a KANJI character.
- The encoding for KSC5601 is the double-byte subset (Code set 1) of EUC-KR, with no check on code points performed by NonStop SQL/MX. For the best results, use the big-endian byte order to denote a KSC5601 character.
- In C/C++ embedded applications, the data type for each KANJI/KSC5601 character is `wchar_t`. Use wide-character C functions instead of single-byte C functions on the host variables.

Fixed-Length Character Data

The C/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` and wide character string handling routines `wcslon` and `wprintf`) require the null terminator. Follow these guidelines for handling the null terminator when you declare and use character arrays as host variables for string literals.

Declaring a Fixed-Length Character Host Variable

When you declare a character array as a host variable, the C/C++ preprocessor reserves the last character of the array as a placeholder for a null terminator. To allow for the extra character, declare a character array one character longer than the actual number of required characters. The `INVOKE` directive automatically appends an extra character to a character array. You can also use the preprocessor option of `-n`, which null terminates host variable character strings before they are fetched into.

Example

This example uses a declaration for an SQL column up to 20 characters in length:

C

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

Selecting Character Data

In a C/C++ program, when selecting character data from a database to return to a host variable array, NonStop SQL/MX 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.

Example

A database contains a PRODUCTS table that consists of the PROD_NUM and PROD_DESC columns. The product number is defined to be the primary key. This example selects character data and appends a null terminator to the hv_prod_desc array before printing the data:

```
C EXEC SQL BEGIN DECLARE SECTION;
      unsigned NUMERIC (4)   find_num;
      unsigned NUMERIC (4)   hv_prod_num;
      char                  hv_prod_desc[11];
EXEC SQL END DECLARE SECTION;

...
EXEC SQL
      SELECT prod_num, prod_desc INTO :hv_prod_num, :hv_prod_desc
      FROM products WHERE prod_num = :find_num;

...
/* append null terminator before displaying string */
hv_prod_desc[10] = '\0';
printf("%hu %s\n", hv_prod_num, hv_prod_desc);
```

Inserting or Updating Fixed-Length Character Data

Fixed-length character columns should always be padded with blanks. If the number of characters in an array (not including the null terminator) is less than the size of the character column, you must pad the array with blanks before inserting it into the database. Otherwise, in a C program, if the number of characters is less than the size of the character column, the INSERT statement stores the null terminator in the database, and comparison operations fail.

Examples

This example inserts data into the PRODUCTS table. The hv_prod_desc array is six characters long (five characters for the column value and one character for the null terminator). Five characters are to be inserted into the prod_desc column:

```
C EXEC SQL BEGIN DECLARE SECTION;
      unsigned NUMERIC (4) hv_prod_num;
      char hv_prod_desc[6]; /* use for a 5-character column */
EXEC SQL END DECLARE SECTION;

...
memcpy(hv_prod_desc, "abc  ", 5); /* copy 5 characters      */
                                   /* (abc plus 2 blanks)    */
hv_prod_desc[5]='\0';

...
EXEC SQL INSERT INTO products (prod_num, prod_desc)
```

```
VALUES (:hv_prod_num, :hv_prod_desc);
```

```
...
```

This example pads the `hv_prod_desc` array with blanks before inserting the array into the database and shows another way to initialize a string host variable:

```
C /* Function to pad an array of characters with */
/* blanks on the right and add null terminator. */

void blank_pad(char *buf, size_t size);
...
int main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) hv_prod_num;
    char hv_prod_desc[11];
    EXEC SQL END DECLARE SECTION;
    ...
    /* Initialize to blank first */
    strncpy(hv_prod_desc, " ", sizeof(hv_prod_desc));
    /* Then copy the initial value in */
    strcpy(hv_prod_desc, "abc"); /* Copy 3 characters */
    blank_pad(hv_prod_desc, sizeof(hv_prod_desc) - 1);
    EXEC SQL INSERT INTO products (prod_num, prod_desc)
        VALUES (:hv_prod_num, :hv_prod_desc);
    ...
} /* end main */

void blank_pad(char *buf, size_t size)
{
    size_t i;
    i = strlen(buf);
    if (i < size)
        memset(&buf[i], ' ', size - i);
    buf[size] = '\0';
} /* end blank_pad */
```

Note. If you are using MXCI, be sure to blank pad fixed-length character arrays, even when inserting or updating a null value. If you do not blank pad the array, selected data in MXCI shows misaligned data. However, the selected data in the embedded SQL program appears correctly.

Variable-Length Character Data

In a C/C++ program, you can retrieve from and insert character data into an SQL VARCHAR column in the same way you do for a `char` column—by declaring a host variable as a fixed-length character array. You can also declare a VARCHAR host variable in an embedded SQL C/C++ program.

Declaring a VARCHAR Host Variable

In addition to the C `char` data type, SQL-99 provides the VARCHAR data type for host variables within embedded SQL C/C++ programs. For host variables declared as the VARCHAR data type, the SQL C/C++ preprocessor generates a C `char` data type.

When you declare a VARCHAR array as a host variable, the SQL C/C++ preprocessor by default reserves the last character of the array as a placeholder for a null terminator. Therefore, declare a VARCHAR array one character longer than the actual number of required characters.

Example

This example uses a declaration for an SQL column up to 20 characters in length:

C

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

Follow the guidelines outlined in [Fixed-Length Character Data](#) on page 3-17 for handling the null terminator when you declare and use VARCHAR arrays as host variables for variable-length string literals.

Using the VARCHAR compatible structure to hold VARCHAR data

In addition to the VARCHAR data type, you can use the following structure to store VARCHAR data. You can use the structure as the input and the output host variable instead of the `varchar` variable:

```
Struct{
    short len;
    char val[size];
}<struct name>;
```

where:

`len` is a numeric data item that represents the length.

`val` is a fixed-length character data item for the string.

The following is a declaration of the VARCHAR compatible structure:

```
EXEC SQL BEGIN DECLARE SECTION;

struct
{
short len;
char val[20];
}test1;

EXEC SQL END DECLARE SECTION;
```

The following examples use the VARCHAR compatible structure as a host variable in the SELECT, INSERT, and UPDATE statements:

- SELECT


```
EXEC SQL SELECT name INTO :test1 FROM t1;
```
- INSERT


```
test1.len = 3;
Strcpy(test1.val, "abc");
EXEC SQL INSERT INTO t1 VALUES(:test1);
```
- UPDATE


```
EXEC SQL UPDATE t1 SET name = :test1 WHERE name = 'xyz';
```

Generating Structures Instead of Using Null-Terminated Strings

Prior to SQL/MX Release 1.8, all C/C++ VARCHAR columns were interpreted as null-terminated strings. Beginning with SQL/MX Release 1.8, NonStop SQL/MX implemented the -a preprocessor option, which translates VARCHAR host variables into structures that contain the correct length and string fields. This behavior is similar to invoked VARCHAR, which is generated as a structure with a length followed by a string. If a VARCHAR structure is also a rowset, an additional array specification is part of the structure definition.

In C/C++, the default behavior of the VARCHAR host variable type is:

```
VARCHAR hvar[size+1];
```

This host variable is translated to:

```
char hvar[size+1];
```

When you specify the -a option, the preprocessor generates this structure:

```
struct{
    short len;
```

```
char val[size+1];
} hvar;
```

Inserting or Updating Variable-Length Character Data

The rules for selecting and inserting or updating variable-length character data are similar to the rules for fixed-length character data with one exception. When inserting or updating data with VARCHAR data type, you do not have to pad the array with blanks.

Example: Using a Null-Terminated String

Using the SQL/MX default for VARCHAR (for example, null terminated string), this example uses a VARCHAR declaration for an SQL column up to 11 characters in length:

```
C EXEC SQL BEGIN DECLARE SECTION;
      unsigned NUMERIC (4) hv_prod_num;
      VARCHAR          hv_prod_desc[11];
EXEC SQL END DECLARE SECTION;

...
strcpy(hv_prod_desc, "abc"); /* Copy 3 characters */
hv_prod_desc[3]='\0';
...
EXEC SQL INSERT INTO products (prod_num, prod_desc)
      VALUES (:hv_prod_num, :hv_prod_desc);
...
```

In contrast to `char` data, for VARCHAR data, you do not need to insert blanks following the data up to the null terminator.

Example: Using a Structure

This example is the same as the previous one except that the preprocessor option `-a` is used. The preprocessor `-a` option specifies that C/C++ interpret VARCHAR host variables as structures and generate structures that contain the correct length and string fields. For details, see [Generating Structures Instead of Using Null-Terminated Strings](#) on page 3-21.

If you use the `-a` option, you must specify the value (`val`) and length (`len`) of the structure when assigning data to the host variable.

```
C EXEC SQL BEGIN DECLARE SECTION;
      unsigned NUMERIC (4) hv_prod_num;
      VARCHAR          hv_prod_desc[11];
EXEC SQL END DECLARE SECTION;

...
strncpy(hv_prod_desc.val, "abc", 3); /* Copy 3 characters */
hv_prod_desc.len = 3;
...
EXEC SQL INSERT INTO products (prod_num, prod_desc)
```

```
VALUES (:hv_prod_num, :hv_prod_desc);
...
```

Note. If the preprocessor -a option is used, the -n preprocessor option has no effect on VARCHARs.

Numeric Data

Use the NUMERIC data type for fixed-point numeric data, the DECIMAL data type for ASCII numeric data, and the PICTURE 9's data type for either fixed-point (COMP) or ASCII (DISPLAY) numeric data.

Assigning SQL Numeric Data to C Character Arrays

Any of the SQL numeric data types can be assigned to a host variable with `char` data type in a C program by first performing the appropriate conversion.

When you use `char` arrays as host variables for NUMERIC, DECIMAL, or PICTURE 9's data, use the SQL/MX CAST function to convert the value:

- If you are converting NUMERIC, PICTURE 9's, or DECIMAL data to a C `char` array and the scale is zero, declare the `char` array two characters larger than the number of digits you expect to store in the array. The first character is the sign (+, -, or blank), and the last character is reserved for the null terminator. If the scale is greater than zero, declare the `char` array three characters larger than the number of digits. The extra character is reserved for the decimal point.
- Within a SELECT INTO or FETCH statement, use the SQL/MX CAST function in the select list to convert the numeric value from the database to the CHAR data type.
- Append a null terminator to the output character string before you process it as a C character string.

See the CAST Specification in the *SQL/MX Reference Manual*.

Initializing DECIMAL Data Types

When initializing the DECIMAL data type, use this `strcpy` command, where `host_var` is defined as DECIMAL:

```
strcpy(host_var, " 83445589");
```

Because the host variable is defined as DECIMAL (8,5), the length is 10 (8+2, minus one character for the null terminator, which makes it 9).

Initializing NUMERIC Data Types

Initialize NUMERIC data types in your application with an assignment statement. No `strcpy` or `CAST` is needed. NUMERIC types are binary types. In this example, the value 10 is moved to `host_var`, which is declared as a `long`.

```
EXEC SQL BEGIN DECLARE SECTION;
NUMERIC(7,0) host_var;
EXEC SQL END DECLARE SECTION;

host_var=10;
```

Initializing BigNum Data Types

You can initialize the BigNum data type with a constant `int64` value, either from another BigNum value or from an ASCII string that represents a BigNum value.

BigNum Data Type

A BigNum value is an array of `int16` elements. The last element that contains the sign is the most significant bit. The BigNum data type is defined as follows:

```
typedef Int16* BignumExt
```

BigNum Functions

You can use the following BigNum functions to perform basic operations on a BigNum Host variable:

- [Assignment Functions](#)
- [Comparison Functions](#)
- [Conversion Functions](#)
- [Arithmetic Functions](#)

Assignment Functions

You can assign an `Int64` value to a BigNum host variable or a BigNum value to an `Int64` variable using the assignment functions.

SQL_BigNumAssign Function

The `SQL_BigNumAssign` function assigns an `Int64` value to a BigNum host variable:

```
int SQL_BigNumAssignI64( BignumExt result, UInt16 resultLen,
Int64 valOp)
```

The `SQL_BigNumAssign` function sets the value of `result` from `valOp`.

If the assignment is successful, this function returns 0. Otherwise, the function returns an `EXE_NUMERIC_OVERFLOW` error, if the `resultLen` is not large enough to hold the `valOp` data.

SQL_BigNumI64Assign

The SQL_BigNumI64Assign function assigns a BigNum value to an Int64 variable:

```
int SQL_BignumI64Assign( Int64 *result, Bignum valOp,
                        UInt16 valOpLen )
```

The SQL_BigNumI64Assign function sets the value of `result` from `valOp`.

If the assignment is successful, this function returns 0. Otherwise, the function returns an EXE_NUMERIC_OVERFLOW error.

Comparison Functions

You can compare two BigNum values by using the comparison functions.

SQL_BigNumCmp

SQL_BigNumCmp function compares two BigNum values:

```
int SQL_BignumCmp( Int16 *result, BignumExt operand1,
                  UInt16 operand1Len, BignumExt operand2, UInt16 operand2Len)
```

Sets the `result` to:

- 0, if `operand1` is equal to `operand2`
- 1, if `operand1` is greater than `operand2`
- 1, if `operand1` is less than `operand2`

If the comparison is successful, this function returns 0, otherwise, the function returns -1.

Conversion Functions

You can convert a BigNum value to an ASCII character string or convert an ASCII character string to a BigNum value, or find the number of bytes required for a BigNum value, if given a precision, using the conversion functions.

SQL_BignumFromStr

The SQL_BignumFromStr function converts an ASCII string to a BigNum value:

```
int SQL_BignumFromStr( BignumExt result, UInt16 resultLen,
                      char const *str)
```

This function converts the `str` operand to a BigNum value and stores the value in the `result` parameter.

If the conversion is successful, this function returns 0. Otherwise, the function returns an EXE_NUMERIC_OVERFLOW error.

SQL_BignumToStr

The `SQL_BignumToStr` function converts a `BigNum` value to an ASCII string representation.

```
int SQL_BignumToStr( char * result, UInt16 resultLen,
BignumExt bnValue, UInt16 bnValueLen)
```

Converts the `BigNum` `bnValue` into its ASCII string representation in the `result`.

If the conversion is successful, this function returns 0. Otherwise, the function returns an `EXE_STRING_OVERFLOW` error.

SQL_BignumSize

The `SQL_BignumSize` converts a precision to the required number of bytes to store a `BigNum` value:

```
int SQL_BignumSize( UInt16 precision)
```

This function returns the number of bytes that is required to hold a `BigNum` value for the given `BigNum` `precision`.

Arithmetic Functions

You can perform arithmetic operations on `BigNum` values and retrieve the sign of a `BigNum` value using the arithmetic functions.

SQL_BignumAdd

The `SQL_BignumAdd` function adds two `BigNum` values:

```
int SQL_BignumAdd( BignumExt result, BignumExt operand1,
BignumExt operand2, UInt16 len )
```

The `result` is the sum of `operand1` and `operand2`. Both the operands must have the same length and scale.

If the operation is successful, this function returns 0. Otherwise, the function returns an `EXE_NUMERIC_OVERFLOW` error.

SQL_BignumSub

The `SQL_BignumSub` function subtracts a `BigNum` value from the other:

```
int SQL_BignumSub( BignumExt result, BignumExt operand1,
BignumExt operand2, UInt16 len )
```

The `result` is the difference between `operand1` and `operand2`. `operand2` is subtracted from `operand1`. Both operands must have the same length and scale.

If the operation is successful, this function returns 0. Otherwise the function returns an `EXE_NUMERIC_OVERFLOW` error.

SQL_BignumMul

The `SQL_BignumMul` function multiplies two `BigNum` values:

```
int  SQL_BignumMul(  BignumExt  result, UInt16 resultLen,
BignumExt operand1, UInt16 operand1Len, BignumExt operand2,
UInt16 operand2Len)
```

The `result` is the product of `operand1` and `operand2`.

If the operation is successful, this function returns 0. Otherwise, the function returns an `EXE_NUMERIC_OVERFLOW` error.

SQL_BignumDiv

The `SQL_BignumDiv` function divides two `BigNum` values.

```
int  SQL_BignumDiv(  BignumExt result, UInt16 resultLen,
BignumExt operand1, UInt16 operand1Len, BignumExt operand2,
UInt16  operand2Len)
```

The `result` is the quotient of dividing `operand1` by `operand2`.

The function returns:

- 0, if the division does not produce a remainder
- 1, if the division produces a remainder
- `EXE_DIVISION_BY_ZERO`, if the `operand2` value is zero.
- `EXE_NUMERIC_OVERFLOW` error, if the division is not successful

SQL_BignumSign

The `SQL_BignumSign` function retrieves the sign of a `BigNum` value:

```
void  SQL_BignumSign(  Int16 * sign, BignumExt  value, UInt16
valueLen )
```

The `sign` operand is set to 0 for positive `BigNum` values and -1 for negative `BigNum` values.

Considerations for BigNum Arithmetic function

When you declare a `BigNum` host variable (`NUMERIC (20, 3)`), `SQL/MX` translates the value to a short data type array in the C language. The short data type array has no scale. The scale is implicit and must be handled in the application. For example, you want to assign the value 1234 to the `BigNum` host variable using the `BigNum` assignment functions. If you place the value 1234 directly, the decimal point is implicit. Therefore, the actual value interpreted by the `SQL/MX` is 1.273. To place the value 1273 into the table, the value 1237000 must be assigned to the `BigNum` assignment functions.

Precision, Magnitude, and Scale of BigNum Arithmetic Results

Precision is the maximum number of digits in the BigNum host variable. *Magnitude* is the number of digits to the left of the decimal point. *Scale* is the number of digits to the right of the decimal point.

For example, if a host variable is declared as NUMERIC (128, 5), the *Precision* is 128, the *Magnitude* is 123, and the *Scale* is 5.

- If the operator is addition (+) or subtraction (-),
 - The resulting *precision* is the maximum of the magnitudes of the operands, plus the scale of the result, plus 1.
- If the operator is multiplication (*),
 - The resulting *scale* is the sum of the scales of the operands.
 - The resulting *precision* is the total of the sum of the magnitudes of the operands and the scale of the result.
- If the operator is division (/),
 - The resulting *scale* is the sum of the scale of the numerator and the magnitude of the denominator.
 - The resulting *magnitude* is the sum of the magnitude of the numerator and the scale of the denominator.
 - The resulting *precision* is the sum of the *magnitude* of the result and the *scale* of the result.

For example, if the numerator is NUMERIC (70, 10) and the denominator is NUMERIC (50, 20), the *precision* and the *scale* of the result will be calculated as follows:

Magnitude of the numerator is $70 - 10 = 60$.

Scale of the numerator is 10.

Magnitude of the denominator is $50 - 20 = 30$.

Scale of the denominator is 20.

Therefore, *Scale* of quotient = $10 + 30 = 40$.

Therefore, *Magnitude* of quotient = $60 + 20 = 80$.

Hence, *precision* of the quotient = $80 + 40 = 120$.

GNU GMP library for BigNum

The user program uses the GNU GMP library for extensive number processing. You can convert the new BigNum external binary format into the GMP format by using GMP

low-level library routines, `mpz_import()`. The following example shows the usage of `mpz_import()` function:

```
// Convert the str which has the value "12345678901234567890"
// from internal format to GMP.

//Declare the Bignum host variable
    NUMERIC(50,5) s;
//Get the storageLength for the NUMERIC s
    unsigned short storageLength =SQL_BignumSize(50);
//First, convert the str "12345678901234567890' to Internal
//format
        SQL_BignumFromStr(s, storageLength , str,
        (unsignedshort)strlen(str));
//Declare the Bignum in GMP format
mpz_t z;
// Convert the Internal format s to GMP format z
mpz_import(z,      // Output multi-precision in GMP format.
           5,      // Number of chunks in s.
           -1,     // Least significant chunk first.
           2,      // Size of each chunk, in bytes.
           0,      // Native endianness within the chunk.
           0,      // Nails.
           s        // Input Bignum in internal format.
           );
```

You can choose to convert a GMP multi-precision value into the new BigNum external format by using GMP low-level library routines `mpz_import()`. The following example shows the usage of `mpz_import()` function:

```
// Convert 12345678901234567890 from GMP to Bignum
// external format.
//Declare the Bignum in GMP format
    mpz_t z;
// Will contain the internal format.
    NUMERIC(50,5) s;          size_t count;
//initialize the bignum in GMP format
mpz_init_set_str(z, "12345678901234567890", 10);
// Convert the GMP format z to internal format s
mpz_export(s,          // Output Bignum in internal format.
           &count,     // The number of chunks written to s.
           -1,         // Least significant chunk first.
           2,          // Size of each chunk, in bytes.
           0,          // Native endianness within the chunk.
           0,          // Nails. I do not know what this is.
           z            // Input Bignum in GMP format.
           );
```

BigNum Format for TMFARLIB

The BigNum data type is displayed using a *presentation type* by TMFARLIB2. The *presentation type* for a BigNum type value in TMFARLIB2 is specified by the following Guardian DEFINE:

```
Add Define =_MX_ARLIB_BIGNUM_FORMAT, CLASS=MAP
FILE=\$dummy.dummy.value
```

value can be an ASCII or a BINARY value.

This is a MAP define. The node name, volume, and subvolume parts are ignored, only the file name part of the DEFINE value is interpreted by SQL/MX. Valid values for the filename part are as follows:

- BINARY - The BigNum binary format is used as the *presentation type*. This is default.
- ASCII - The ASCII format is used as the *presentation type*.

For example, the DEFINE in the binary format is as follows:

```
add_define =_MX_ARLIB_BIGNUM_FORMAT class=MAP
file=\$dummy.dummy.BINARY
```

If the DEFINE is set to a value which is not valid, it is ignored and the binary format is used as the *presentation type*.

Storing C Character Arrays Into SQL Numeric Columns

You can use C character arrays as host variables in a C program when inserting or updating values into numeric columns.

When you use `char` arrays as host variables for NUMERIC, DECIMAL, or PICTURE 9's columns, use the SQL/MX CAST function to convert the value:

- Format a C character string, without the null terminator, consisting of a sign, digits, and a decimal point. The formatted C character string must be left-justified and padded with blanks within the character array.
- Within an INSERT or UPDATE statement, use the SQL/MX CAST function to convert the character data to the desired SQL data type. The character set string can be associated with either ISO88591 or UCS2 character set.

Assigning Numeric Data to Corresponding Data Types

You can perform C arithmetic operations on SQL columns of NUMERIC, PICTURE 9's, or DECIMAL data type. To do so, assign the data to host variables with the same data type as the database data type (that is, NUMERIC, PICTURE 9's, or DECIMAL) as shown in [Table 3-1](#) on page 3-9.

[Table 3-5](#) lists the extended SQL C data types NUMERIC and PICTURE 9's COMP. Use NUMERIC or PICTURE 9's COMP host variables within your C program as shown.

Table 3-5. Host Variable Usage for NUMERIC or PICTURE 9's COMP Data		
C Program Usage for 32-bit Address Model	C Program Usage for 64-bit Address Model	NUMERIC or PICTURE 9's COMP Data
short	short	The length <i>l</i> (the precision) is 1 through 4 digits.
long	int	The length <i>l</i> (the precision) is 5 through 9 digits.
long long	long long	The length <i>l</i> (the precision) is 10 through 18 digits.

[Table 3-6](#) also lists the extended SQL C data types DECIMAL and PICTURE 9's DISPLAY. Use DECIMAL or PICTURE 9's DISPLAY host variables within your C program as shown in [Table 3-6](#).

Table 3-6. Host Variable Usage for DECIMAL or PICTURE 9's DISPLAY Data

C Program Usage	DECIMAL or PICTURE 9's DISPLAY Data
<code>char[l + 2]</code>	The length <code>l</code> is the precision of the numeric data. Two extra characters are allocated by the preprocessor: the first character to store the sign (+, −, or blank) and the last character to store the null terminator for the character string.

Assigning Fixed-Point Data Types

If you assign fixed-point values, an SQL NUMERIC data type with scale, 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/MX 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/MX stores the integral part of the value and generates a warning to indicate a loss of data (the fractional part). Use this assignment only when you intend to truncate the fractional part.
- When you declare a fixed-point value (NUMERIC), NonStop SQL/MX translates the value to 'long long' in the C language. The 'long long' data type has no scale. In the next example, when the value is assigned using host variable `binary_64_s`, the host variable value is 1273. The decimal point is implicit, so the actual value interpreted by SQL is 1.273. To place a value of 1273 into the table, the application must use `binary_64_s=1237000`. The scale is implicit and must be handled in the application.

```
int main()
{
    Exec sql set catalog 'ework';
    Exec sql set schema 'ework';

    SQLCODE = -1;
    memset(SQLSTATE, 32, 6);

    strncpy(insert_buf, " ", sizeof(insert_buf));
    strcpy(insert_buf, "INSERT INTO tb32 (binary_64_s) VALUES
        (cast(? as numeric(18, 3)));");

    EXEC SQL PREPARE ins FROM :insert_buf;
    printf("SQLCODE after PREPARE ins is %d\n", SQLCODE);

    binary_64_s = 1273;

    ia = 0;

    EXEC SQL EXECUTE ins USING :binary_64_s indicator :ia;
```

```
printf("SQLCODE after insert - 3 is %d\n", SQLCODE);

EXEC SQL COMMIT WORK;
printf("\nSQLCODE after COMMIT WORK is %d\n\n", SQLCODE);

return 0;
```

Assigning Floating-Point Data Types

A column in an SQL/MX format table declared with a floating-point data type is stored in IEEE floating-point format, and all computations on it are done assuming that. In addition, all computations performed in MXCI, either on SQL/MX or SQL/MP tables, use IEEE floating-point format.

A column in an SQL/MP format table declared with a floating-point data type is stored in Tandem floating-point format, and all computations on it are done assuming that. NonStop SQL/MX provides an option to store SQL/MP floating-point data in host variables in IEEE floating-point format. The difference between IEEE and Tandem floating-point format is seen in the storage, precision, and implementation for the formats. Input and output via host variables can be treated as IEEE or Tandem floating-point format.

If you use NonStop SQL/MX to perform computations on a floating-point column in an SQL/MP table, the result is in IEEE format.

For dynamic queries in embedded SQL programs, the default is Tandem floating-point format.

For static queries in embedded SQL programs, the input and output depend on the type of host variables that are declared. This table summarizes the use of IEEE and Tandem floating points for host variables:

Embedded SQL/C Host Variable	Is the -o option used during compilation?	Translated C/C++ Declaration for Host Variables	Generated Definition in Module Definition File
float	No*	float (Tandem)	REAL
double	No*	double (Tandem)	DOUBLE PRECISION
float	Yes**	float (IEEE)	REAL_IEEE
double	Yes**	double (IEEE)	DOUBLE_IEEE

* Default for TNS/R-targeted compilations

** Default for TNS/E-targeted compilations

Restrictions for Floating-Point Format

- You cannot declare both IEEE and Tandem floating-point host variables in an application program.
- You cannot use the IEEE floating-point format on hardware that does not support the format.

- The conversion between Tandem floating point and IEEE floating point in an application is not transparent.

Changing the Format of Floating-Point Data Types

To change the format of floating-point data types (FLOAT, REAL, or DOUBLE PRECISION data types) for dynamic SELECT statements and dynamic parameters, use the CONTROL QUERY DEFAULT setting, FLOATTYPE, which is Tandem floating point by default. For information on CONTROL QUERY DEFAULT settings, see the *SQL/MX Reference Manual*.

The SQL/MX `-o` preprocessor option directs the preprocessor to generate IEEE floating-point types instead of Tandem floating-point types for host variables. For information on preprocessor settings, see [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

Conversion Between Tandem and IEEE Floating-Point Formats

When converting a Tandem floating-point data type to the corresponding IEEE data type, all Tandem floating-point data types are converted to IEEE DOUBLE representation. Despite this conversion, the precision of Tandem floating-point data types are maintained correctly in IEEE floating-point format. A Tandem REAL or FLOAT with precision between 1 and 22 cannot be converted to IEEE REAL because the Tandem exponent will not fit in an IEEE REAL data type. Although no equivalent exists for single-precision Tandem REAL and FLOAT in IEEE floating-point format, the conversion to IEEE DOUBLE preserves the precision and the exponent.

If you want a small floating-point data type with a smaller exponent and less storage, consider declaring the host variables as `float`. If you want more exponent and a larger precision, consider declaring the host variables as `double` or `float(p)`.

For more information on floating-point formats, see the *SQL/MX Reference Manual*.

Date-Time and Interval Data

Use the following for date-time and interval data types:

DATE	Represents a date.
TIME	Represents a time.
TIMESTAMP	Represents a timestamp.
INTERVAL	Represents a duration of time as a year-month or day-time interval.

For SQL/MP DATETIME data types that are not equivalent to DATE, TIME, or TIMESTAMP, you are still required to declare a character array host variable and use the CAST function for input to and output from date-time or interval columns, similar to SQL/MX Release 1.8, which does not support ANSI-99 date-time host variables.

DATE Representation

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 09/15/1993, 1993-09-15, or 15.09.1993. You control the display format by inserting the value in the format you want and retrieving the value by using the DATEFORMAT function. See the DATEFORMAT function in the *SQL/MX Reference Manual*.

For example, if a table in the database has this column definition:

HIRE_DATE DATE

The host variable representation for December 22, 1988, in DEFAULT format is:

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

A DATE host variable in DEFAULT format is represented as an 11-character string, including 10 characters—with hyphens (-) as field separators—plus a character (empty space) for a null terminator.

Selecting Standard Date-Time Values

To retrieve standard date-time values (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) from the database, declare a date-time (DATE, TIME, or TIMESTAMP) host variable. For the required number of digits for DATE, TIME, or TIMESTAMP values, see [Table 3-3](#) on page 3-11.

If your C program performs string operations on the date-time host variable, you must append a null terminator to the output string before processing it because the date-time data types are internally processed as C character strings.

[Table 3-7](#) lists the lengths of the target arrays for TIME and TIMESTAMP values, which depend on the precision (the number of digits in the fractional seconds).

Table 3-7. Lengths of C Target Arrays for TIME and TIMESTAMP

TIME Precision	Length	TIMESTAMP Precision	Length
TIME	9	TIMESTAMP	27
TIME(0)	9	TIMESTAMP(0)	20
TIME(1)	11	TIMESTAMP(1)	22
TIME(2)	12	TIMESTAMP(2)	23
TIME(3)	13	TIMESTAMP(3)	24
TIME(4)	14	TIMESTAMP(4)	25
TIME(5)	15	TIMESTAMP(5)	26
TIME(6)	16	TIMESTAMP(6)	27

The TIME default precision is 0 (zero), and the TIMESTAMP default precision is 6.

Example

If a database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns, this example selects the date-time value:

```
C EXEC SQL BEGIN DECLARE SECTION;
      struct billing_rec {
          unsigned short  hv_custnum;
          DATE            hv_billing_date;
          ...
      } bill;
      ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT custnum, billing_date
      INTO :bill.hv_custnum, :bill.hv_billing_date
      FROM billings
      WHERE custnum = :hv_this_customer;
... bill.hv_billing_date[10]='\0';
```

Inserting or Updating Standard Date-Time Values

To insert or update standard date-time values (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) in the database, format the date-time values in the desired display format for a date, time, or timestamp. Within an INSERT or UPDATE statement, use the DATE, TIME, or TIMESTAMP data type.

Example

If a database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns, this example inserts a customer number and date-time value into that table:

```
C EXEC SQL BEGIN DECLARE SECTION;
      struct billing_rec {
          unsigned short  hv_custnum;
          DATE            hv_billing_date;
          ...
      } bill;
      ...
EXEC SQL END DECLARE SECTION;
...
      bill.hv_billing_date[10]='\0';
...
EXEC SQL INSERT INTO billings
      VALUES (:bill.hv_custnum, :bill.hv_billing_date);
...
```


Selecting SQL/MP DATETIME Values Not Equivalent to DATE, TIME, or TIMESTAMP

To retrieve nonstandard SQL/MP DATETIME values that are not equivalent to DATE, TIME, or TIMESTAMP, declare a C char array one character larger than the number of characters you expect to store in the array. For a list of nonstandard SQL/MP DATETIME data types, see the *SQL/MX Reference Manual*.

Use the SQL/MX CAST function to convert a date-time column in a select list to a character string. You must also specify the length in the AS clause of the CAST function to be the length of the declared host variable minus 1. Append a null terminator to the output character string before you process it as a C character string.

Example

Suppose that an SQL/MP database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns. The BILLING_DATE column has a DATETIME MONTH TO DAY data type, which has no equivalent in SQL/MX. This example selects the SQL/MP DATETIME value:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    struct billing_rec {
        unsigned short  hv_custnum;
        char             hv_billing_date[6];
    } bill;
    ...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT custnum, CAST(billing_date AS CHAR(5))
    INTO :bill.hv_custnum, :bill.hv_billing_date
    FROM billings
    WHERE custnum = :hv_this_customer;
... bill.hv_billing_date[5]='\0';
```

Inserting or Updating SQL/MP DATETIME Values Not Equivalent to DATE, TIME, or TIMESTAMP

To insert or update nonstandard SQL/MP DATETIME values that are not equivalent to DATE, TIME, or TIMESTAMP, format a C character string in the desired display format for a date, time, or timestamp. Within an INSERT or UPDATE statement, use the SQL/MX CAST function to convert the character date-time data to a DATE, TIME, or TIMESTAMP data type.

If you are using date-time values as input values to the database in statements other than INSERT or UPDATE (for example, within the WHERE clause of a SELECT statement), you must also use the CAST function to convert the character string to a DATE, TIME, or TIMESTAMP data type.

Example

Suppose that an SQL/MP database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns. The BILLING_DATE column has a DATETIME MONTH TO DAY data type, which has no equivalent in SQL/MX. This example inserts a customer number and date-time value into that table:

```

C EXEC SQL BEGIN DECLARE SECTION;
    struct billing_rec {
        unsigned short  hv_custnum;
        char            hv_billing_date[6];
    } bill;
EXEC SQL END DECLARE SECTION;
...
    bill.hv_billing_date[5]='\0';
EXEC SQL INSERT INTO billings
    VALUES (:bill.hv_custnum,
            CAST(:bill.hv_billing_date AS DATETIME MONTH TO DAY));
...

```

INTERVAL Representation

Interval values are represented as character strings, with a separator between the values of the fields (year-month or day-time). An extra character is generated at the beginning of the interval string for a sign.

For example, if a table in the database has this column definition:

```
AGE    INTERVAL YEAR(2) TO MONTH
```

The host variable representation for 36 years, 7 months, is:

Sign	Year	Separator	Month	Null
+	36	-	07	

An INTERVAL host variable is represented as a seven-character string, including five characters—with a hyphen (-) as the field separator—plus a character for the sign and a character (empty space) for a null terminator.

Selecting Interval Values

To retrieve interval values from the database, declare an INTERVAL host variable the same length as the number of bytes you expect to store in the array. The SQL/MX preprocessor adds two extra characters to the interval string—one for the sign and one (an empty space) for a null terminator.

If your C program performs string operations on the interval host variable, you must append a null terminator to the output string before processing it because the interval data type is internally processed as a C character string.

Example

A database contains a BILLINGS table consisting of the CUSTNUM, START_DATE, BILLING_DATE, and TIME_BEFORE_PMT columns. This example selects a customer number and interval value:

```

C EXEC SQL BEGIN DECLARE SECTION;
      struct billing_rec {
        unsigned short  hv_custnum;
        DATE            hv_start_date;
        DATE            hv_billing_date;
        INTERVAL DAY(3) hv_time_before_pmt;
      } bill;
      ...
EXEC SQL END DECLARE SECTION;
...

EXEC SQL SELECT custnum, time_before_pmt
  INTO :bill.hv_custnum, :bill.hv_time_before_pmt
  FROM billings
  WHERE custnum = :hv_this_customer;
... bill.hv_time_before_pmt[4]='\0';

```

Inserting or Updating Interval Values

To insert or update interval values, format a C interval string in the desired display format for an interval. The first character is reserved for the sign of the interval.

Example

A database contains a BILLINGS table consisting of the CUSTNUM, START_DATE, BILLING_DATE, and TIME_BEFORE_PMT columns. This example updates date-time and interval values:

```

C EXEC SQL BEGIN DECLARE SECTION;
      struct billing_rec {
        unsigned short  hv_custnum;
        DATE            hv_start_date;
        DATE            hv_billing_date;
        INTERVAL DAY(3) hv_time_before_pmt;
      } bill;
      ...
EXEC SQL END DECLARE SECTION;
...
      bill.hv_start_date[10]='\0'
      bill.hv_billing_date[10]='\0';

/* Blank pad if the interval data is less than 3 digits. */
...
strcpy(bill.hv_time_before_pmt, " 111");
EXEC SQL UPDATE billings
  SET billing_date = :bill.hv_billing_date,
      time_before_pmt = :bill.hv_time_before_pmt
  WHERE custnum = :hv_this_customer;
...

```

If the interval string length is larger than the number of digits in the interval value, you must blank pad up to the null terminator. See [Inserting or Updating Fixed-Length Character Data](#) on page 3-18.

Host Variables in C Structures

When you refer to a single field name in a structure, you must include the structure name with the field name.

Example

This example uses a structure named `employee_info`, containing the `empnum` and `empname` fields:

```
C EXEC SQL BEGIN DECLARE SECTION;
struct employee {                /* structure definition */
    unsigned short empnum;
    char empname[21];
};
struct employee emp_input;       /* directly allocated structure */
struct employee * emp_output;    /* structure pointer */
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 names:

```
emp_output = malloc(sizeof(employee));
emp_input.empnum = 100;

EXEC SQL SELECT empname
-- reference hostvar through struct pointer
INTO :emp_output->empname
FROM CAT.SCH.EMPLOYEE
-- reference hostvar as struct field
WHERE empnum = :emp_input.empnum;
```

Host Variables as Data Members of a C++ Class

You can include an SQL Declare Section within a class definition to use a data member of a class as a host variable.

Example

This example uses a class named `jobsql`, containing the declarations of the `memhv_jobcode` and `memhv_jobdesc` host variables. These host variables are referenced in the member function `putjob` defined in the class:

```
C++ class jobsql {
    // Class member host variables
    EXEC SQL BEGIN DECLARE SECTION;
        unsigned NUMERIC (4) memhv_jobcode;
        VARCHAR          memhv_jobdesc[19];
    EXEC SQL END DECLARE SECTION;
```

```

public:
void putjob(){
    EXEC SQL
        INSERT INTO persnl.job
            VALUES (:memhv_jobcode, :memhv_jobdesc);
}
}; // End of jobsql class definition
main(){
    ...
    jobsql mysql; // Instantiate a member of the class jobsql
    // Insert job code in table
    mysql.putjob();
} // End of main

```

Using Indicator Variables in a C/C++ Program

Null in an SQL column indicates that a value is either unknown or is not applicable. A host language program uses an indicator variable to insert null. It also uses an indicator variable to test for null or a truncated output value.

An indicator variable is an exact numeric variable associated with the host variable that sets or receives the actual column value. The INVOKE directive automatically declares indicator variables for columns that allow null.

A host language program can use an indicator variable to:

- Insert values into a database with an INSERT or UPDATE statement.
- Test for null or a truncated value (in the case of character data) after retrieving a value from a database with a SELECT INTO or FETCH statement.

For more information on indicator variables, see [Specifying Host Variables in SQL Statements](#) on page 3-15.

Inserting Null

To insert values into columns that allow null with an INSERT or UPDATE statement, you must set the indicator variable to a value less than zero for null or zero for a nonnull value before executing the statement.

Example

This example uses a statement that inserts values into the ODETAIL table. The columns UNIT_PRICE and QTY_ORDERED allow null. To insert null, declare and use an indicator variable:

C

```

EXEC SQL BEGIN DECLARE SECTION;
...
    short ind_1 = -1;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL INSERT INTO sales.odetail
    (ordernum, partnum, unit_price, qty_ordered)

```

```
VALUES ( :hv_ordernum, :hv_partnum,
         :hv_unit_price :ind_1,
         :hv_qty_ordered :ind_1 );
```

Testing for Null or a Truncated Value

To test for null or a truncated character value, check the indicator variable associated with a host variable. If the value of the indicator variable is less than zero, the associated column contains null. If the value of the indicator variable is greater than zero, character data in the column was truncated when it was assigned to the host variable.

Example

This example selects values from the PARTS table and returns these values to host variables. The columns PARTDESC and QTY_AVAILABLE allow null. After the SELECT statement executes, the example tests the indicator variable for null or a truncated value:

C

```
...
EXEC SQL SELECT partnum, partdesc, price, qty_available
        INTO :hv_partnum,
            :hv_partdesc
            INDICATOR :hv_partdesc_i,
            :hv_price,
            :hv_qty_available
            INDICATOR :hv_qty_available_i,
FROM sales.parts
WHERE partnum = :in_partnum;
...
if (hv_qty_available_i < 0) handle_null_value();
if (hv_partdesc_i > 0 ) handle_truncated_value();
...
```

Retrieving Rows With Nulls

To retrieve a row that contains null, use the NULL predicate in the WHERE clause. You cannot use an indicator variable set to -1 in a WHERE clause to retrieve a row that contains null. If you do, NonStop SQL/MX does not find the row and returns a NOTFOUND exception even if a column actually contains null.

Example

This example retrieves rows that have nulls from the EMPLOYEE table using a cursor. The cursor specifies the NULL predicate in the WHERE clause in the associated SELECT statement:

C

```
/* 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 :hv_empnum,
           :hv_first_name,
           :hv_last_name,
           :hv_deptnum,
           :hv_jobcode,
           :hv_salary ;
/* Process the row that contains the null salary. */
/* Branch back to FETCH the next row.           */
...
EXEC SQL CLOSE get_null_salary ;
...

```

Creating C Host Variables Using INVOKE

The INVOKE preprocessor directive creates a structure with the names of the host variables corresponding to columns in a table or view. INVOKE converts the column names to C identifiers and generates a C declaration for each column. If a column allows null, INVOKE also creates an indicator variable for the column.

You can declare host variables that correspond to the columns in an SQL table or view without using an INVOKE statement. However, using an INVOKE statement to generate host variables has these advantages:

- **Program independence:** If you modify a table or view, the INVOKE statement re-creates the host variables to correspond to the new table or view when you run the SQL/MX C preprocessor. However, you must modify a program that refers to a deleted column or accesses a new column.
- **Performance:** The INVOKE statement maps SQL data types to the corresponding host language data types, and usually no data conversion is required at run time. For further information, see [Example 3-1](#) on page 3-47 and [Example 3-2](#) on page 3-48.
- **Program readability and maintenance:** The INVOKE statement creates host variables using the same names as column names in the table or view.

Using the INVOKE Directive

To execute an INVOKE directive for a table or view, you must have SELECT privileges on all applicable columns when you run the SQL/MX C preprocessor. The general syntax for using an embedded INVOKE directive within an SQL Declare Section in a C program is:

```

EXEC SQL INVOKE table-or-view [AS structure-name];
struct structure-name structure-instance ;

```

For complete syntax, see the INVOKE Directive in the *SQL/MX Reference Manual*.

The `struct` declaration declares *structure-instance* to be a structure of the type named *structure-name*. You must declare a variable of the `struct` type so that you can use that variable in your C language statements.

If you do not specify the AS clause in the INVOKE statement, the default structure name is the simple name of the table or view with the suffix `_type` appended. For example: `mytable_type`.

INVOKE and Date-Time and Interval Host Variables (SQL/MX Release 1.8 Applications)

SQL/MX Release 1.8 does not support SQL:1999 date-time host variables. In SQL/MX Release 1.8 applications, you must declare a character array host variable for date-time or interval data and use the CAST function for input or output from date-time or interval columns.

If your SQL/MX Release 1.8 application uses INVOKE to create a date-time or interval host variable and you plan to preprocess the application in SQL/MX Release 2.x, use the `-e` preprocessor option. Otherwise, SQL/MX Release 2.x returns an error during SQL compilation because the CAST function in the program is incompatible with the SQL:1999 date-time host variables created by INVOKE in SQL/MX Release 2.x.

Note. This issue affects only SQL/MX Release 1.8 applications preprocessed by SQL/MX Release 2.x. Previously compiled SQL/MX Release 1.8 applications continue to run correctly without changes in SQL/MX Release 2.x.

For more information, see [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

INVOKE and Floating-Point Host Variables

For floating-point columns, INVOKE generates a structure that uses the Tandem or IEEE floating-point format depending on whether the `-o` preprocessor option is used during compilation. [Table 3-8](#) lists the C declarations that INVOKE generates for each type of floating-point column and shows the effect of the `-o` option on the floating-point format and on the execution of the application.

Table 3-8. INVOKE and Floating-Point Host Variables

SQL Column Data Type	Is the <code>-o</code> option used during compilation?	INVOKE-Generated C Declaration	Outcome of Application Execution
REAL SQL/MP column (Tandem format)	No*	<code>float</code> (Tandem)	Successful execution
	Yes**	<code>double</code> (IEEE)	Successful execution
FLOAT (1 to 22 bits) SQL/MP column (Tandem format)	No*	<code>float</code> (Tandem)	Successful execution
	Yes**	<code>double</code> (IEEE)	Successful execution
FLOAT (23 to 54 bits) SQL/MP column (Tandem format)	No*	<code>double</code> (Tandem)	Successful execution
	Yes**	<code>double</code> (IEEE)	Successful execution
DOUBLE PRECISION SQL/MP column (Tandem format)	No*	<code>double</code> (Tandem)	Successful execution
	Yes**	<code>double</code> (IEEE)	Successful execution but precision might be lost
REAL SQL/MX column (IEEE format)	No*	<code>float</code> (Tandem)	Successful execution but precision might be lost
	Yes**	<code>float</code> (IEEE)	Successful execution
FLOAT (1 to 52 bits) SQL/MX column (IEEE format)	No*	<code>double</code> (Tandem)	Overflow or underflow errors might occur.
	Yes**	<code>double</code> (IEEE)	Successful execution
DOUBLE PRECISION SQL/MX column (IEEE format)	No*	<code>double</code> (Tandem)	Overflow or underflow errors might occur.
	Yes**	<code>double</code> (IEEE)	Successful execution

* Default for TNS/R-targeted compilations
 ** Default for TNS/E-targeted compilations

Because the range of the IEEE REAL data type is smaller than the range of the Tandem REAL data type, IEEE REAL host variables cannot accommodate data from SQL/MP REAL columns, which are in Tandem floating-point format. If the `-o` preprocessor option is used with invoked SQL/MP tables that have a column of type REAL, this option causes the invoked structure to be of type DOUBLE, enabling values from SQL/MP REAL columns to fit into an IEEE host variable.

If your application uses INVOKE to generate floating-point host variables for SQL/MX columns, you should use the `-o` option. The `-o` option overrides the use of Tandem floating point and uses IEEE floating point instead.

The `-o` option is particularly important for SQL/MX DOUBLE columns. If the `-o` option is not used, INVOKE generates Tandem `double` host variables for SQL/MX DOUBLE columns, which are in IEEE format. However, the range of IEEE DOUBLE is larger than the range of Tandem DOUBLE, which could cause overflow or underflow errors during the execution of the application. Therefore, verify that the `-o` option is specified when using INVOKE on SQL/MX DOUBLE columns. For information on preprocessor settings, see [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

C Data Types Generated by INVOKE

To show the correspondence between a table named SQLCDATA that contains columns of various SQL data types and the C structure generated by an INVOKE statement, the INVOKE statement and `struct` declaration are coded:

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE sqlcdata AS sqlc_types_struct;
struct sqlc_types_struct sqlc_types;
...
EXEC SQL END DECLARE SECTION;
```

The SQL/MX C preprocessor generates the structure immediately after the INVOKE statement in the preprocessed program code.

[Example 3-1](#) and [Example 3-2](#) show the columns for the `sqlcdata` table and the corresponding generated data types in the structure `sqlc_types_struct`.

[Example 3-1](#) shows the CREATE TABLE statement that generates the SQLCADATA table.

Example 3-1. CREATE TABLE Statement

```
CREATE TABLE samdbcat.mysch.sqlcdata (
type_char          CHAR (10)                NOT NULL,
type_char_null     CHAR (10)                ,
type_UCS2_CHAR     CHAR (10) CHARACTER SET UCS2 NOT NULL,
type_UCS2_VARCHAR  VARCHAR (10) CHARACTER SET UCS2 NOT NULL,
type_picx          PIC X(10)                NOT NULL,
type_varchar       VARCHAR (10)             NOT NULL,
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_piccomp4_s    PIC S9(2)V9(2)           COMP    NOT NULL,
type_piccomp4_u    PIC 9(2)V9(2)           COMP    NOT NULL,
type_piccomp9_s    PIC S9(7)V9(2)           COMP    NOT NULL,
type_piccomp9_u    PIC 9(7)V9(2)           COMP    NOT NULL,
type_piccomp18_s   PIC S9(16)V9(2)          COMP    NOT NULL,
type_dec4_s        DECIMAL (4)              SIGNED  NOT NULL,
type_dec4_u        DECIMAL (4)              UNSIGNED NOT NULL,
type_dec9_s        DECIMAL (9,2)            SIGNED  NOT NULL,
type_dec9_u        DECIMAL (9,2)            UNSIGNED NOT NULL,
type_dec18_s       DECIMAL (18,2)           SIGNED  NOT NULL,
type_pic4_s        PIC S9(2)V9(2)           NOT NULL,
type_pic4_u        PIC 9(2)V9(2)           NOT NULL,
type_pic9_s        PIC S9(7)V9(2)           NOT NULL,
type_pic9_u        PIC 9(7)V9(2)           NOT NULL,
type_pic18_s       PIC S9(16)V9(2)          NOT NULL,
type_small_s       SMALLINT                 SIGNED  NOT NULL,
type_small_u       SMALLINT                 UNSIGNED NOT NULL,
type_small_null    SMALLINT                 ,
type_int_s         INTEGER                  SIGNED  NOT NULL,
type_int_u         INTEGER                  UNSIGNED NOT NULL,
type_large_s       LARGEINT                 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_date          DATE                     NOT NULL,
type_time_6        TIME                     NOT NULL,
type_timestamp_6   TIMESTAMP                NOT NULL,
type_interval      INTERVAL YEAR TO MONTH  NOT NULL
) ;
```

[Example 3-2](#) shows the structure generated by this INVOKE directive (with CHAR AS STRING):

```
EXEC SQL INVOKE sqlcdata AS sqlc_types_struct;
struct sqlc_types_struct sqlc_types;
```

Example 3-2. C Structure Generated by INVOKE

```
/* Beginning of generated code for SQL INVOKE */
struct sqlc_types_struct{
    long long syskey;
    char /* CHARACTER SET ISO88591 */ type_char[11];
    short type_char_null_i;
    char /* CHARACTER SET ISO88591 */ type_char_null[11];
    wchar_t /* CHARACTER SET UCS2 */ type_char_ucs2[11];
    wchar_t /* CHARACTER SET UCS2 */ type_varchar_ucs2[11];
    char /* CHARACTER SET ISO88591 */ type_picx[11];
    char /* CHARACTER SET ISO88591 */ type_varchar[11];
    short          type_num4_s;
    unsigned short type_num4_u;
    long           type_num9_s;
    unsigned       long type_num9_u;
    long long      type_num18_s;
    short          type_piccomp4_s;
    unsigned short type_piccomp4_u;
    long           type_piccomp9_s;
    unsigned long  type_piccomp9_u;
    long long      type_piccompl8_s;
    char           type_dec4_s[6];
    char           type_dec4_u[6];
    char           type_dec9_s[11];
    char           type_dec9_u[11];
    char           type_dec18_s[20];
    char           type_pic4_s[6];
    char           type_pic4_u[6];
    char           type_pic9_s[11];
    char           type_pic9_u[11];
    char           type_pic18_s[20];
    short          type_small_s;
    unsigned short type_small_u;
    short          type_small_null_i;
    short          type_small_null;
    long           type_int_s;
    unsigned long  type_int_u;
    long long      type_large_s;
    float          type_float_15;
    double         type_float_30;
    float          type_real;
    double         type_dbl_prec;
    char           type_date[11];
    char           type_time_6[9];
    char           type_timestamp_6[27];
    char           type_interval[7];
};
```

Using Indicator Variables With the INVOKE Directive

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

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 the INVOKE statement to generate an indicator variable name derived from the column name and the prefix or suffix. A default suffix of `_i` applies if the INVOKE directive omits these clauses.

Example

The table named `c_table` has the columns `empnum` and `empname`. The column `empname` can be null. This example uses an INVOKE statement with both the PREFIX and SUFFIX clauses:

C

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE c_table PREFIX beg_ SUFFIX _end;
struct c_table_type mytable;
...
EXEC SQL END DECLARE SECTION;
```

The SQL C preprocessor generates this structure immediately after the INVOKE directive in the preprocessed program code:

```
/* Beginning of generated code for SQL INVOKE */
struct c_table_type {
    long      empnum;
    short     beg_empname_end;
    char      empname[16];
};
```

In this example, the structure name defaults to the simple name of the table or view with the suffix `_type` appended. The structure name is `c_table_type`.

You must declare a variable of the `struct` type so that you can use that variable in your C language statements and your embedded SQL statements. In this example, the declared `struct` variable is named `mytable`.

NULL STRUCTURE Clause

The NULL STRUCTURE clause causes the INVOKE statement to generate a structure for a column that allows null. It assigns the same name as the column to the structure. The structure includes fields for the data item, named `valu`, and its indicator variable, named `indicator`.

Example

A database contains an EMPTBL table consisting of the columns EMPNUM, FIRST_NAME, LAST_NAME, and HIRE_DATE. The columns FIRST_NAME and HIRE_DATE allow null. This example uses an INVOKE statement with the NULL STRUCTURE clause:

C

```
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE emptbl AS emptbl_rec NULL STRUCTURE;
struct emptbl_rec emptbl_rec1, emptbl_rec2;
...
EXEC SQL END DECLARE SECTION;
```

The SQL/MX C preprocessor generates this structure template immediately after the INVOKE statement in the preprocessed program code:

```
/* Beginning of generated code for SQL INVOKE */
struct emptbl_rec {
    unsigned short empnum;
    struct {
        short indicator;
        char valu[16];
    } first_name;
    char last_name[21];
    struct {
        short indicator;
        char valu[11];
    } hire_date;
};
```

The SQL/MX C preprocessor supplies only the structure template. You must supply the variable declarations for this struct type of the form:

```
struct emptbl_rec emptbl_rec1, emptbl_rec2;
```

Your program code would then include statements with host variables of the form:

```
...
EXEC SQL OPEN get_emptbl_rec;
...
EXEC SQL FETCH get_emptbl_rec INTO
:emptbl_rec1.empnum,
:emptbl_rec1.first_name.valu
INDICATOR :emptbl_rec1.first_name.indicator,
:emptbl_rec1.last_name,
:emptbl_rec1.hire_date.valu
INDICATOR :emptbl_rec1.hire_date.indicator;
...
```

For columns that allow null, the target host variable names begin with the appropriate struct variable, followed by the particular null structure struct variable, and end with either the indicator or the valu within the generated null structure.

C Example of Using INVOKE

The next example declares and uses host variable names and indicator variable names and shows:

- A host variable declaration using INVOKE that includes indicator variables. The structure is declared as global so that any function can reference the host variables.
- A host variable indicator variable used in the SELECT statement. The columns that might contain null require the indicator variable following the host variable to receive information on nulls.
- An indicator variable testing for null. If the value of the indicator variable following the SELECT is less than zero, the associated column is null.

[Example 3-3](#) retrieves four columns of an order detail table. The table is like the ODETAIL table of the sample database except that the UNIT_PRICE and QTY_ORDERED columns allow null.

Example 3-3. C INVOKE (page 1 of 2)

C

```
...
void handle_null(void);
void display_result(void);
...
EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL INVOKE samdbcat.mysch.odetail;
/* Beginning of generated code for SQL INVOKE */
struct odetail_type {
    unsigned long    ordernum;
    unsigned short   partnum;
    short            unit_price_i;
    long             unit_price;
    short            qty_ordered_i;
    unsigned long    qty_ordered;
} ;
struct odetail_type odetail_rec;
EXEC SQL END DECLARE SECTION;

int main()
{
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (6) in_ordernum;
    unsigned NUMERIC (4) in_partnum;
EXEC SQL END DECLARE SECTION;
...

```

Example 3-3. C INVOKE (page 2 of 2)

```

/* Initialize the host variables in the WHERE clause. */
...
EXEC SQL
  SELECT ordernum, partnum, unit_price, qty_ordered
  INTO :odetail_rec.ordernum,
       :odetail_rec.partnum,
       :odetail_rec.unit_price
       INDICATOR :odetail_rec.unit_price_i,
       :odetail_rec.qty_ordered
       INDICATOR :odetail_rec.qty_ordered_i
  FROM sales.odetail
  WHERE ordernum = :in_ordernum AND partnum = :in_partnum;
...
if (odetail_rec.unit_price_i < 0 ||
    odetail_rec.qty_ordered_i < 0)
  handle_null();
else
  display_result();

return 0;
} /* end main */

void handle_null()
{
  ...
} /* end handle_null */

void display_result()
{
  printf("Order number: %6d\n", odetail_rec.ordernum);
  printf("Part number: %4d\n", odetail_rec.partnum);
  printf("Unit price: %10.2lf\n", odetail_rec.unit_price/100.);
  printf("Quantity ordered: %5d\n", odetail_rec.qty_ordered);
} /* end display_result */

```

Character Set Examples

This set of examples shows methods for selecting, fetching, and inserting data using different character sets. The examples are based on these tables:

```

CREATE TABLE STAFF_UC
  (EMPNUM   CHAR(3) character set ucs2 NOT NULL UNIQUE,
   EMPNAME  NCHAR VARYING(20),
   GRADE    DECIMAL(4),
   CITY     VARCHAR(15) character set ucs2);

CREATE TABLE PROJ
  (PNUM     CHAR(3) NOT NULL,
   PNAME    VARCHAR(20),
   PTYPE    CHAR(7),
   BUDGET   DECIMAL(9),
   CITY     VARCHAR(15),

```



```
    primary key(pnum)
);
```

Selecting From a UCS2 Character Set Into a VARCHAR Host Variable

This example selects from a UCS2 character set into a VARCHAR host variable:

```
EXEC SQL WHENEVER SQLERROR CALL handle_error;

//select VARCHAR ucs2 column to VARCHAR UCS2 host variables with
the same length

void select_varchar2varchar_ucs2()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char CHARACTER SET IS UCS2 hv_empnum[4];
        VARCHAR CHARACTER SET UCS2 hv_empname[21];
        VARCHAR CHARACTER SET UCS2 hv_city[16 CHARACTERS];
    EXEC SQL END DECLARE SECTION;

    hv_empnum[3] = '\0';

    EXEC SQL
        select empnum,empname,city into :hv_empnum, :hv_empname,
            :hv_city
        from staff_uc
        where empnum < _ucs2'E8' and grade = 10;

    //use the value in hv_empnum, hv_empname, and hv_city
}
```

Fetching From a UCS2 Character Set into a VARCHAR Host Variable

This example uses a FETCH operation from a UCS2 character set into a VARCHAR host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
    long SQLCODE;
    static char SQLSTATE_OK[6] = "00000";
    char SQLSTATE[6];
    char CHARACTER SET IS UCS2 hv_empnum[4];
    VARCHAR CHARACTER SET UCS2 hv_empname[21];
    DECIMAL(4) hv_grade;
    VARCHAR CHARACTER SET UCS2 hv_city[16 CHARACTERS];
EXEC SQL END DECLARE SECTION;

//select UCS2 columns into host variable
void fetch_varchar2varchar_ucs2()
{
    EXEC SQL BEGIN DECLARE SECTION;
```

```

    int hv_cnt = 0;
EXEC SQL END DECLARE SECTION;

hv_empnum[3] = '\0';

EXEC SQL
    declare curs01 cursor for
        select * from staff_uc order by empnum;

EXEC SQL open curs01;

EXEC SQL fetch curs01 into :hv_empnum, :hv_empname,
    :hv_grade, :hv_city;
while (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
{
    // process the output values :hv_empnum, :hv_empname,
    // :hv_grade, and :hv_city

EXEC SQL FETCH curs01 INTO
    :hv_empnum, :hv_empname, :hv_grade, :hv_city;
}
}

```

Selecting From an ISO88591 Character Set Into a UCS2 Host Variable

This example uses the host variable `relaxation` with selection from an ISO88591 character set into a UCS2 host variable and comparing an ISO88591 column with a UCS2 host variable in the WHERE clause:

```

EXEC SQL BEGIN DECLARE SECTION;
    long SQLCODE;
    static char SQLSTATE_OK[6] = "00000";
    char SQLSTATE[6];
    char CHARACTER SET UCS2 out_hv_pnum[4];
    char CHARACTER SET UCS2 out_hv_ptype[8];
    VARCHAR CHARACTER SET UCS2 out_hv_pname[21];
    VARCHAR CHARACTER SET UCS2 out_hv_city[16];
    VARCHAR CHARACTER SET UCS2 in_hv_city[16];
    char CHARACTER SET UCS2 in_hv_city2[16];
    int hv_cnt;
EXEC SQL END DECLARE SECTION;

//varchar ISO88591 column compare with varchar UCS2 hostvar
void relaxation_example_comparison()
{
    //init input hostvar
    wcsncpy((wchar_t *)in_hv_city, L"Cupertino");
    out_hv_ptype[7] = (wchar_t)'\0';

EXEC SQL
    declare curs02 cursor for
        select pnum, pname, ptype from proj

```

```
        where city <> :in_hv_city
        order by pnum;

EXEC SQL open curs02;

EXEC SQL fetch curs02 INTO
        :out_hv_pnum, :out_hv_pname, :out_hv_ptype;

while (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
{
    // processthe output

    EXEC SQL fetch curs02 into
        :out_hv_pnum, :out_hv_pname, :out_hv_ptype;
    cout <<"-----"<<endl;
}
EXEC SQL close curs02;
}
```


Host Variables in COBOL Programs

Host variables are data items declared in a host application program and used in both host language statements and embedded SQL statements. They provide communication between SQL statements and the host language statements. An input host variable transfers data from a host language program to an SQL/MX database, and an output host variable transfers data from a database to the program.

This section describes:

- [Specifying a Declare Section](#) on page 4-1
- [COBOL Host Variable Data Types](#) on page 4-2
- [Using Corresponding SQL and COBOL Data Types](#) on page 4-5
- [Specifying Host Variables in SQL Statements](#) on page 4-9
- [Using Host Variables in a COBOL Program](#) on page 4-10
- [Using COBOL Data Description Clauses](#) on page 4-18
- [Using Indicator Variables in a COBOL Program](#) on page 4-19
- [Creating COBOL Host Variables Using INVOKE](#) on page 4-22
- [Character Set Examples](#) on page 4-30

Specifying a Declare Section

You declare all host variables within an SQL Declare Section. When you specify a Declare Section:

- Use the BEGIN DECLARE SECTION statement to begin a Declare Section.
- Use the END DECLARE SECTION statement to end a Declare Section.
- The first item after BEGIN DECLARE SECTION must have level 01 or level 77.
- For the best performance, declare the host variable as the same data type as the column in the SELECT list. If you declare this way, you can use bulk moves to input and output data.
- You can specify more than one Declare Section in your source file, but do not nest them.
- Do not place a Declare Section within a COBOL record description.
- Do not place any executable code within a Declare Section.
- You can use SQL or host language comments in a Declare Section.

Example

This example uses host variable declarations in an SQL Declare Section:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 CUST-REC.
    02 CUSTNUM      PIC S9(4) COMP.
```

```

02 CITY          PIC X(14).
...
EXEC SQL INVOKE SALES.PARTS AS SALES-REC END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.

```

COBOL Host Variable Data Types

You must explicitly declare all host variables used in SQL statements. A host variable used in an SQL statement must be declared in an SQL Declare Section prior to the first use of the host variable in an SQL statement. Only variables of the types recognized by the 3GL preprocessor can appear within an SQL Declare Section.

To declare a COBOL host variable, specify one of these data types:

```

[char-set]PIC[TURE][IS] {X[(length)]}
| nn column-name.
  mm LEN   PIC S9(4) COMP.
  mm VAL   [char-set] PIC X(1).
| DATE
| TIME [(n)]
| TIMESTAMP [(n)]
| INTERVAL [period1[(n)] | SECOND [(n[,m)]]) [TO
  period2[(m)]]
| PIC[TURE] [S]9(1-s)V9(s) COMP
| PIC[TURE] [S]9(p) COMP
| PIC[TURE] S9(1-s)V9(s) DISPLAY SIGN LEADING SEPARATE
| PIC[TURE] 9(1-s)V9(s) DISPLAY

```

[*char-set*]PIC[TURE] X(1)

specifies the data type of a target host variable for a column of one of these SQL data types:

```

CHAR[ACTER] [(1)]
PIC[TURE] X(1)

```

The length 1 corresponds to the length of the column value.

[*char-set*] can be specified as:

```

CHARACTER SET [IS] character-set-name

```

The optional [*char-set*] clause specifies the character set to be associated with the host variable. *character-set-name* can be ISO88591 or UCS2 for SQL/MX tables. *character-set-name* can be KANJI or KSC5601 for SQL/MP tables. The default character set is ISO88591. Note that you can use host variables with the KANJI or KSC5601 character set in an SQL/MX application only to access KANJI or KSC5601 columns in an SQL/MP table.

```

nn  column-name .
    mm LEN  PIC S9(4) COMP.
    mm VAL  [char-set] PIC X(1)

```

specifies the data type of a target host variable for a column of the SQL data type:

VARCHAR(1)

The length 1 corresponds to the maximum length of the column value. The level numbers are indicated by *nn* and *mm*. The level number *nn* can be any level in the range 01 to 49, where *mm* is a greater level than *nn*. LEN specifies the actual length of the character item in VAL. VAL is a character data item with length 1, specifying the maximum number of characters that can be stored in VAL.

DATE

specifies the data type of a target host variable for a date-time column that contains a date in the external form *yyyy-mm-dd*.

TIME [(n)]

specifies the data type of a target host variable for a date-time column that, without the optional *n* precision, contains a time in the external form *hh:mm:ss*. The *n* precision is a positive integer that specifies the number of digits in the fractional seconds. The default for the precision is 0, and the maximum is 6.

TIMESTAMP [(n)]

specifies the data type of target host variable for a date-time column that, without the optional *n* precision, contains a timestamp in the external form:

yyyy-mm-dd hh:mm:ss

The *n* precision is a positive integer that specifies the number of digits in the fractional seconds, as shown in bold text:

yyyy-mm-dd hh:mm:ss.msssss

The default for precision is 6, and the maximum is 6.

INTERVAL [*period1*[(*n*)] | SECOND [(*n*[,*m*)]]) [TO *period2*[(*m*)]]

where

n leading precision

m fractional precision

period YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

period 1 must be greater or equal time part than *period 2*. YEAR to SECOND is valid. SECOND to YEAR is invalid.

Specifies a column that represents a duration of time as either a year-month or day-time range or a single-field. *period1* can have a leading-precision up to 18 digits (the maximum depends on the number of fields in the interval). The leading-

precision is the number of digits allowed in *period 1*. If *period2* is SECOND, it can have a fractional-precision up to 6 digits. The fractional-precision is the number of digits of precision after the decimal point. The default for leading-precision is 2, and the default for fractional-precision is 6. If the single-field is SECOND, the leading-precision is the number of digits of precision before the decimal point, and the fractional-precision is the number of digits of precision after the decimal point.

```
PIC[TURE] [S]9(1-s)V9(s) COMP
```

specifies the data type of a target host variable for a column of one of these SQL data types:

```
NUMERIC [(p, s)] [SIGNED|UNSIGNED]
PIC[TURE] [S] {9(1-s) V9(s) | V9(s)} COMP
```

p precision of the column value
s scale of the column value
1 length; number of digits in the column value
1-s number of digits in the integral part of the column value

```
PIC[TURE] [S]9(p) COMP
```

specifies the data type of a target host variable for a column of one of these SQL data types, depending on the precision:

```
SMALLINT [SIGNED|UNSIGNED]
INT[EGER] [SIGNED|UNSIGNED]
LARGEINT
```

The precision *p* corresponds to the precision of the column value.

```
PIC[TURE] S9(1-s)V9(s) DISPLAY SIGN LEADING SEPARATE
```

specifies the data type of a target host variable for a column of one of these SQL data types:

```
DECIMAL (1, s) SIGNED
PIC[TURE] S9(1-s) V9(s) DISPLAY SIGN IS LEADING
```

1 length; number of digits in the column value
s scale of the column value
1-s number of digits in the integral part of the column value


```
PIC[TURE] 9(1-s)V9(s) DISPLAY
```

specifies the data type of a target host variable for a column of one of these SQL data types:

```
DECIMAL (1, s) UNSIGNED
PIC[TURE] 9(1-s) V9(s) DISPLAY
```

l length; number of digits in the column value

s scale of the column value

1-s number of digits in the integral part of the column value

For the corresponding SQL and COBOL host variable data types, see [Table 4-1](#).

Using Corresponding SQL and COBOL Data Types

[Table 4-1](#) lists the embedded SQL COBOL host variable “a[100]” with all its legal SQL/MX modifiers, the equivalent data type in NonStop SQL/MX, and the translated COBOL declarations.

Table 4-1. COBOL Character Host Variables and Their SQL Data Type Equivalents and COBOL Translations (page 1 of 2)

COBOL Host Variable	SQL/MX Equivalent Data Type	Translated COBOL Declaration
A PIC X(100) A PIC X (100 CHARACTERS)	CHAR(100) CHARACTER SET ISO88591	A PIC X(100)
A CHARACTER SET IS ISO88591 PIC X(100) A CHARACTER SET IS ISO88591 PIC X (100 CHARACTERS)	CHAR(100) CHARACTER SET ISO88591	A PIC X(100)
A CHARACTER SET IS UCS2 PIC X(100) A CHARACTER SET IS UCS2 PIC X(100 CHARACTERS)	CHAR(100) CHARACTER SET UCS2	A PIC X(200)
A CHARACTER SET IS KANJI ¹ PIC X(100) A CHARACTER SET IS KANJI ¹ PIC X(100 CHARACTERS)	CHAR(100) CHARACTER SET KANJI ¹	A PIC X(200)
A CHARACTER SET IS KSC5601 ¹ PIC X(100) A CHARACTER SET IS KSC5601 ¹ PIC X(100 CHARACTERS)	CHAR(100) CHARACTER SET KSC5601 ¹	A PIC X(200)

Table 4-1. COBOL Character Host Variables and Their SQL Data Type Equivalents and COBOL Translations (page 2 of 2)

COBOL Host Variable	SQL/MX Equivalent Data Type	Translated COBOL Declaration
01 <i>column-name</i> . 03 LEN PIC S9(4) COMP. 03 VAL [<i>char-set</i>] PIC X(1).	CHAR [ACTER] VARYING(1) VARCHAR [ACTER](1)	01 <i>column-name</i> . 03 LEN PIC S9(4) COMP. 03 VAL [<i>char-set</i>] PIC X(1).
DATE	DATE	PIC[TURE] X(1). ²
TIME [(n)]	TIME [(<i>time-precision</i>)]	PIC[TURE] X(1). ²
TIMESTAMP[(n)]	TIMESTAMP[(<i>time-precision</i>)]	PIC[TURE] X(1). ²
INTERVAL [<i>period1</i> [(n)] SECOND [(n[,m])]] [TO <i>period2</i> [(m)]]	INTERVAL { <i>start-field</i> TO <i>end-field</i> <i>single-field</i> }	PIC[TURE] X(1). ³
¹ A positive integer that represents the length. ¹ KANJI and KSC5601 character sets can be used only with SQL/MP tables. ² For DATE, the value of the length <i>l</i> is 10. For TIME(6), the value of the length <i>l</i> is 15. For TIMESTAMP(6), the value of the length <i>l</i> is 26. ³ The INTERVAL data type has an extra character for a sign. The sign is included in the length <i>l</i> .		

[Table 4-2](#) lists the corresponding SQL data type, C host variable data type, and translated COBOL declaration for the NUMERIC, DECIMAL, PIC, SMALLINT, LARGEINT, and date-time data types. You can specify a COBOL host variable if it has a corresponding SQL data type.

Table 4-2. Corresponding SQL, COBOL Host Variable Data Types, and Translated COBOL Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, LARGEINT, and Date-Time Data Types (page 1 of 2)

SQL Data Type	COBOL Host Variable Data Type	Translated COBOL Declaration
NUMERIC (1 to 4, <i>s</i>) SIGNED	PIC[TURE] S9(4- <i>s</i>)V9(<i>s</i>) COMP. ¹	PIC[TURE] S9(4- <i>s</i>)V9(<i>s</i>) COMP. ¹
NUMERIC (1 to 4, <i>s</i>) UNSIGNED	PIC[TURE] 9(4- <i>s</i>)V9(<i>s</i>) COMP. ¹	PIC[TURE] 9(4- <i>s</i>)V9(<i>s</i>) COMP. ¹
NUMERIC (5 to 9, <i>s</i>) SIGNED	PIC[TURE] S9(9- <i>s</i>)V9(<i>s</i>) COMP. ¹	PIC[TURE] S9(9- <i>s</i>)V9(<i>s</i>) COMP. ⁴
NUMERIC (5 to 9, <i>s</i>) UNSIGNED	PIC[TURE] 9(9- <i>s</i>)V9(<i>s</i>) COMP. ¹	PIC[TURE] 9(9- <i>s</i>)V9(<i>s</i>) COMP. ¹
NUMERIC (10 to 18, <i>s</i>) SIGNED	PIC[TURE] S9(18- <i>s</i>)V9(<i>s</i>) COMP. ¹	PIC[TURE] S9(18- <i>s</i>)V9(<i>s</i>) COMP. ¹
PIC[TURE] [S]9(1- <i>s</i>)V9(<i>s</i>) COMP	Same as NUMERIC	Same as NUMERIC
DEC[IMAL] (1, <i>s</i>) SIGNED	PIC[TURE] S9(1- <i>s</i>)V9(<i>s</i>) DISPLAY SIGN LEADING SEPARATE.	PIC[TURE] S9(1- <i>s</i>)V9(<i>s</i>) DISPLAY SIGN LEADING SEPARATE.
DEC[IMAL] (1, <i>s</i>) UNSIGNED	PIC[TURE] 9(1- <i>s</i>)V9(<i>s</i>) DISPLAY.	PIC[TURE] 9(1- <i>s</i>)V9(<i>s</i>) DISPLAY.
PIC[TURE] [S]9(1- <i>s</i>)V9(<i>s</i>)	Same as DECIMAL	Same as DECIMAL
SMALLINT SIGNED	PIC[TURE] S9(4) COMP. ¹	PIC[TURE] S9(4) COMP. ¹
SMALLINT UNSIGNED	PIC[TURE] 9(4) COMP. ¹	PIC[TURE] 9(4) COMP. ¹
INT[EGER] SIGNED	PIC[TURE] S9(9) COMP. ¹	PIC[TURE] S9(9) COMP. ¹
INT[EGER] UNSIGNED	PIC[TURE] 9(9) COMP. ¹	PIC[TURE] 9(9) COMP. ¹

¹ A positive integer that represents the length.

s A positive integer that represents the scale of the number.

¹ NonStop SQL/MX treats BINARY as COMP[UTATIONAL].

² For DATE, the value of the length *l* is 10. For TIME(6), the value of the length *l* is 15. For TIMESTAMP(6), the value of the length *l* is 26.

³ The INTERVAL data type has an extra character for a sign. The sign is included in the length *l*.

Table 4-2. Corresponding SQL, COBOL Host Variable Data Types, and Translated COBOL Declarations for NUMERIC, DECIMAL, PIC, SMALLINT, LARGEINT, and Date-Time Data Types (page 2 of 2)

SQL Data Type	COBOL Host Variable Data Type	Translated COBOL Declaration
LARGEINT	PIC[TURE] S9(18) COMP. ¹	PIC[TURE] S9(18) COMP. ¹
FLOAT (1 to 22 bits) REAL	Not supported.	Not supported.
FLOAT (23 to 54 bits) DOUBLE PRECISION	Not supported.	Not supported.
DATE	DATE	PIC[TURE] X(1). ²
TIME [(n)]	TIME [(time-precision)]	PIC[TURE] X(1). ²
TIMESTAMP[(n)]	TIMESTAMP[(time-precision)]	PIC[TURE] X(1). ²
INTERVAL [period1[(n)] SECOND [(n[,m])]] [TO period2[(m)]]	INTERVAL {start-field TO end-field single-field }	PIC[TURE] X(1). ³

¹ A positive integer that represents the length.

^s A positive integer that represents the scale of the number.

¹ NonStop SQL/MX treats BINARY as COMP[UTATIONAL].

² For DATE, the value of the length *l* is 10. For TIME(6), the value of the length *l* is 15. For TIMESTAMP(6), the value of the length *l* is 26.

³ The INTERVAL data type has an extra character for a sign. The sign is included in the length *l*.

Data Conversion

NonStop SQL/MX performs the conversion between SQL and COBOL data types:

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

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

Converting Numeric Types

Values of data types NUMERIC, DECIMAL, PICTURE 9's, INTEGER, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION are numbers and are all mutually comparable and mutually assignable.

NonStop SQL/MX converts data between signed and unsigned numeric types and between numeric types with different precision.

If assignment would result in a loss of significant digits, NonStop SQL/MX returns a data exception condition in SQLSTATE. See [Table 13-1](#) on page 13-2.

Converting Character Types

Values of data types CHARACTER, PICTURE X's, and CHARACTER VARYING are character strings and are all mutually comparable and mutually assignable if both are of the same character set. In addition, UCS2 host variables are mutually comparable and assignable with ISO88591 nonhost variable objects.

For character strings of different lengths, NonStop SQL/MX pads the receiving string variable on the right with blanks as necessary.

If the receiving string variable is too short, NonStop SQL/MX truncates the right part of the string retrieved from the database and returns a data exception condition in SQLSTATE. See [Table 13-1](#) on page 13-2.

Specifying Host Variables in SQL Statements

Use COBOL naming conventions for host variable names. A COBOL name can contain from 1 to 30 alphanumeric characters, including letters, digits, and hyphens (-). The first or last letter cannot be a hyphen. Letters can be uppercase, lowercase, or a combination of both.

To use a COBOL record description as a host variable, specify the record name as a level 01 entry and use level numbers 01 to 49 and 77 for the host variables. The individual data items, and not the record name, are the host variables.

After you declare a host variable, to specify it within an embedded SQL statement, use:

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

variable-name

is the host variable name. It can be any valid host language identifier with a data type that corresponds to an SQL data type. You must precede *variable-name* with a colon (:) within an SQL statement.

{OF | IN} *record-name*

is an optional clause that specifies a level 01 item. The *variable-name* or *indicator_variable* must be qualified by the record or group name only if the host variable name or indicator variable name is not unique within the program. This clause is an SQL/MX extension.

INDICATOR

is a keyword that can precede *indicator_variable*.

indicator_variable

is an indicator variable of exact numeric data type. You must declare the indicator field as type `PIC S9(4) COMP` in COBOL. You must precede *indicator_variable* with a colon (:) in an SQL statement. You must declare an indicator variable along with its corresponding host variable within an SQL Declare Section.

If data returned in the host variable is null, the indicator variable is less than zero. If character data returned is truncated, the indicator variable is set to the length of the string in the database. Otherwise, the value of the indicator variable is zero. To insert null into the database, set the indicator variable to a value less than zero.

Using Host Variables in a COBOL Program

As a COBOL programmer, you need to know how to declare and use host variables to retrieve and insert data with these SQL data types:

- [Fixed-Length Character Data](#) on page 4-11
- [Variable-Length Character Data](#) on page 4-12
- [Numeric Data](#) on page 4-12
- [Date-Time and Interval Data](#) on page 4-13

Character Set Data

These guidelines apply for NonStop SQL/MX Release 1.8 and NonStop SQL/MX Release 2.x character sets:

- ISO88591 character set: An SQL/MX Release 1.8 application can be run under SQL/MX Release 2.x without application recompilation, if the application contains ISO88591 character data only.
- KANJI and KSC5601 character set: If KANJI or KSC5601 character set host variables are contained in the application, the application must be carefully rewritten and recompiled. KANJI and KSC5601 host variables in C applications are translated as single-byte arrays in SQL/MX Release 1.8 and as double-byte arrays in SQL/MX Release 2.x. If the application is not rewritten, SQL errors might be emitted, corruption of data might occur, and the application might crash.

Host variable source code for storing KANJI characters in SQL/MX Release 1.8:

```
A PIC X(100).
```

Host variable source code for storing KANJI characters in SQL/MX Release 2.x:

```
A CHARACTER SET KANJI PIC X(100).
```

Guidelines for Revising KANJI/KSC5601 Character Set Host Variables

Follow these guidelines when rewriting an application that contains KANJI or KSC5601 character sets for SQL/MX Release 2.x:

- Use the character set clause `CHARACTER SET IS KANJI` or `CHARACTER SET IS KSC5601`.
- The encoding for KANJI is the double-byte subset of the Shift-JIS, with no check on code points performed by NonStop SQL/MX. For the best results, use the big-endian byte order to denote a KANJI character.
- The encoding for KSC5601 is the double-byte subset (Code set 1) of EUC_KR, with no check on code points performed by NonStop SQL/MX. For the best results, use the big-endian byte order to denote a KSC5601 character.
- In COBOL embedded applications, each KANJI/KSC5601 character is represented by two single-byte characters. When you copy KANJI/KSC5601 objects, always use the correct number of bytes to ensure the entire object is moved.

Embedded COBOL Applications With UCS2 Literals

Because COBOL only understands single-byte character types, the byte order matters when UCS2 literals are encoded in the application. Because the targeted execution machine (NonStop system) is a big-endian machine, the byte order of each UCS2 character entered should be big-endian. For example, the UCS2 character -U+2021 (the double dagger sign) should be coded:

```
move X"20" & x"21" to host-variable.
```

Fixed-Length Character Data

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

```
PIC[TURE] X (length)
```

The *length* value must be a positive integer and not greater than 4096. Instead of *length*, you can specify multiple Xs, with each X representing one character position.

Variable-Length Character Data

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

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

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

Example

The EMPLOYEE table has the LAST_NAME column defined as VARCHAR(20). In a COBOL program, this column is specified as:

```
COBOL 05 HV-LAST-NAME.
      10  LEN  PIC S9(4) COMP.
      10  VAL  PIC X(20).
```

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

```
...
MOVE "SMITH" TO VAL OF HV-LAST-NAME.
MOVE 5 to LEN OF HV-LAST-NAME.
EXEC SQL UPDATE persnl.employee
      SET last_name = :HV-LAST-NAME
      WHERE empnum = :HV-EMPNUM
END-EXEC.
```

Numeric Data

Use this PICTURE DISPLAY clause to declare a host variable for the SQL DECIMAL and PICTURE 9's DISPLAY data types:

```
PICTURE [S] { 9(integer) [V9(scale)] | V9(scale) }
      [USAGE [IS]] DISPLAY [SIGN [IS] LEADING SEPARATE [CHARACTER]]
```

Use this PICTURE COMP or PICTURE BINARY clause to declare a host variable for the SQL NUMERIC, PICTURE 9's COMP, SMALLINT, LARGEINT, and INTEGER data types:

```
PICTURE [S] { 9(integer) [V9(scale)] | V9(scale) }
      [USAGE [IS]] { COMP[UTATIONAL] | COMP | BINARY }
```


COMP and DISPLAY Data

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

Sign, Number of Digits, and Scale

The *S* specifies a signed variable. If you omit *S*, the variable is unsigned.

The *9(integer)* specifies *integer* number of digits; *integer* must be positive. The *V* designates a decimal position.

The *9(scale)* designates the number of positions to the right of the decimal. The value of *scale* must be a positive integer. If you do not specify *scale*, the value 0 is used.

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

The values of *integer* and *scale* determine the size of the column. The sum of these values cannot exceed 18. There is no default numeric column definition. You must specify either *9(integer)* or *V9(scale)*.

You must ensure that the value limit imposed by the PICTURE clause is valid for the data. Corresponding SQL columns defined as type DECIMAL, NUMERIC, PICTURE 9's COMP, SMALLINT, LARGEINT, and INTEGER data types can accept values as large as the limit determined by the column size in bytes.

Date-Time and Interval Data

Use the following for date-time and interval data types:

DATE	Represents a date.
TIME	Represents a time.
TIMESTAMP	Represents a timestamp.
INTERVAL	Represents a duration of time as a year-month or day-time interval.

For SQL/MP DATETIME data types that are not equivalent to DATE, TIME, or TIMESTAMP, you are still required to declare a character array host variable and use the CAST function for input to and output from date-time or interval columns, similar to SQL/MX Release 1.8, which does not support ANSI-99 date-time host variables.

DATE Representation

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 09/15/1993, 1993-09-15, or 15.09.1993. You control the display format by inserting the value in the

format you want and retrieving the value by using the DATEFORMAT function. See the DATEFORMAT function in the *SQL/MX Reference Manual*.

For example, if a table in the database has this column definition:

HIRE_DATE DATE

The host variable representation for May 28, 1992, in DEFAULT format is:

Year		Separator		Month		Separator		Day	
1	9	9	2	–	0	5	–	2	8

A DATE host variable in DEFAULT format is represented as a 10-character string with hyphens (-) as field separators.

Selecting Standard Date-Time Values

To retrieve standard date-time values (DATE, TIME, or TIMESTAMP, or the SQL/MP equivalents) from the database, declare a date-time (DATE, TIME, or TIMESTAMP) host variable. For the required number of digits for DATE, TIME, or TIMESTAMP values, see [Table 4-2](#) on page 4-7.

[Table 4-3](#) lists the lengths of the target arrays for TIME and TIMESTAMP values, which depend on the precision (the number of digits in the fractional seconds).

Table 4-3. Lengths of Target Arrays for TIME and TIMESTAMP

TIME Precision	Length	TIMESTAMP Precision	Length
TIME	8	TIMESTAMP	26
TIME(0)	8	TIMESTAMP(0)	19
TIME(1)	10	TIMESTAMP(1)	21
TIME(2)	11	TIMESTAMP(2)	22
TIME(3)	12	TIMESTAMP(3)	23
TIME(4)	13	TIMESTAMP(4)	24
TIME(5)	14	TIMESTAMP(5)	25
TIME(6)	15	TIMESTAMP(6)	26

The TIME default precision is 0 (zero), and the TIMESTAMP default precision is 6.

Example

If a database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns, this example selects the date-time value:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.  
...  
01 BILLINGS-REC.  
  02 HV-CUSTNUM           PIC 9(4) COMP.  
  02 HV-BILLING-DATE      DATE.
```

```

...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT custnum, billing_date
      INTO :HV-CUSTNUM, :HV-BILLING-DATE
      FROM billings
      WHERE custnum = :HV-THIS-CUSTOMER
      END-EXEC.
...

```

Inserting or Updating Standard Date-Time Values

To insert or update standard date-time values (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) in the database, format the date-time values in the desired display format for a date, time, or timestamp. Within an INSERT or UPDATE statement, use the DATE, TIME, or TIMESTAMP data type.

Example

If a database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns, this example inserts a customer number and date-time value into that table:

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 BILLINGS-REC.
   02 HV-CUSTNUM          PIC 9(4) COMP.
   02 HV-BILLING-DATE     DATE.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL INSERT INTO billings
      VALUES (:HV-CUSTNUM, :HV-BILLING-DATE)
      END-EXEC.
...

```

Selecting SQL/MP DATETIME Values Not Equivalent to DATE, TIME, or TIMESTAMP

To retrieve nonstandard SQL/MP DATETIME values that are not equivalent to DATE, TIME, or TIMESTAMP, declare a COBOL character array the same length as the number of bytes you expect to store in the array. For a list of nonstandard SQL/MP DATETIME data types, see the *SQL/MX Reference Manual*.

Use the SQL/MX CAST function to convert a date-time column in a select list to a character string. You must also specify the length in the AS clause of the CAST function to be the length of the declared host variable.

Example

Suppose that an SQL/MP database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns. The BILLING_DATE column has a

DATETIME MONTH TO DAY data type, which has no equivalent in SQL/MX. This example selects the SQL/MP DATETIME value:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 BILLINGS-REC.
   02 HV-CUSTNUM          PIC 9(4) COMP.
   02 HV-BILLING-DATE     PIC X(10).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT custnum, CAST(billing_date AS CHAR(10))
      INTO :HV-CUSTNUM, :HV-BILLING-DATE
      FROM billings
      WHERE custnum = :HV-THIS-CUSTOMER
      END-EXEC.
...
```

Inserting or Updating SQL/MP DATETIME Values Not Equivalent to DATE, TIME, or TIMESTAMP

To insert or update nonstandard SQL/MP DATETIME values that are not equivalent to DATE, TIME, or TIMESTAMP, format a COBOL character string in the desired display format for a date, time, or timestamp. Within an INSERT or UPDATE statement, use the SQL/MX CAST function to convert the character date-time data to a DATE, TIME, or TIMESTAMP data type.

If you are using date-time values as input values to the database in statements other than INSERT or UPDATE (for example, within the WHERE clause of a SELECT statement), you must also use the CAST function to convert the character string to a DATE, TIME, or TIMESTAMP data type.

Example

Suppose that an SQL/MP database has a BILLINGS table that consists of the CUSTNUM and BILLING_DATE columns. The BILLING_DATE column has a DATETIME MONTH TO DAY data type, which has no equivalent in SQL/MX. This example inserts a customer number and date-time value into that table:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 BILLINGS-REC.
   02 HV-CUSTNUM          PIC 9(4) COMP.
   02 HV-BILLING-DATE     PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL INSERT INTO billings
      VALUES (:HV-CUSTNUM,
              CAST(:HV-BILLING-DATE AS DATETIME MONTH TO DAY))
      END-EXEC.
...
```

INTERVAL Representation

Interval values are represented as character strings, with a separator between the values of the fields (year-month or day-time). An extra character is generated at the beginning of the interval string for a sign.

For example, a table in the database has this column definition:

```
AGE      INTERVAL YEAR(2) TO MONTH
```

The host variable representation for 37 years, 11 months, is:

Sign	Year	Separator	Month
+	3	7	- 1 1

An INTERVAL host variable is represented as a six-character string, including five characters—with a hyphen (-) as the field separator—plus a character for the sign.

Selecting Interval Values

To retrieve interval values from the database, declare an INTERVAL host variable the same length as the number of bytes you expect to store in the array. The SQL/MX preprocessor adds an extra character for the sign.

Example

A database contains a BILLINGS table consisting of the CUSTNUM, START_DATE, BILLING_DATE, and TIME_BEFORE_PMT columns. This example selects a customer number and interval value:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 BILLINGS-REC.
   02 HV-CUSTNUM          PIC 9(4) COMP.
   02 HV-START-DATE       DATE.
   02 HV-BILLING-DATE     DATE.
   02 HV-TIME-BEFORE-PMT  INTERVAL DAY(3).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT custnum, time_before_pmt
  INTO :HV-CUSTNUM, :HV-TIME-BEFORE-PMT
  FROM billings
  WHERE custnum = :HV-THIS-CUSTOMER
END-EXEC.
...
```

Inserting or Updating Interval Values

To insert or update interval values, format a COBOL interval string in the desired display format for an interval. The first character is reserved for the sign of the interval.

Example

A database contains a BILLINGS table consisting of the CUSTNUM, START_DATE, BILLING_DATE, and TIME_BEFORE_PMT columns. This example updates date-time and interval values:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 BILLINGS-REC.
   02 HV-CUSTNUM           PIC 9(4) COMP.
   02 HV-START-DATE        PIC X(10).
   02 HV-BILLING-DATE       DATE.
   02 HV-TIME-BEFORE-PMT    INTERVAL DAY(3).
EXEC SQL END DECLARE SECTION END-EXEC.

...
EXEC SQL UPDATE billings
SET billing_date = :HV-BILLING-DATE,
    time_before_pmt = :HV-TIME-BEFORE-PMT
WHERE custnum = :HV-CUSTNUM
END-EXEC.
...
```

By default, INTERVAL DAY is 2 digits. Therefore, for the preceding example, declare HV-TIME-BEFORE-PMT to be length 3 characters, adding one character for the sign.

Using COBOL Data Description Clauses

[Table 4-4](#) lists the COBOL data description clauses and their interpretation by NonStop SQL/MX when they are used in host variable declarations. NonStop SQL/MX does not support the COBOL special names option DECIMAL POINT IS COMMA.

Table 4-4. Interpretation of COBOL Data Description Clauses (page 1 of 2)

COBOL Description	Host Variable Interpretation
BLANK	The clause is ignored.
data-name	Any data name is allowed, including an SQL reserved word. Specific hyphenation rules apply.
FILLER	The clause is ignored.
JUSTIFIED	The clause is not allowed. However, it can appear in an entry already being ignored, such as REDEFINES.
level number	Any number is allowed. Entries with the level number 66 or 88 are ignored.
PICTURE	The clause must be consistent with the PICTURE clause rules for host variables.
REDEFINES	The clause is ignored.
SIGN	For DISPLAY items, the SIGN clause must be LEADING SEPARATE. No restrictions apply, and the appropriate conversion for SQL data types is made.

Table 4-4. Interpretation of COBOL Data Description Clauses (page 2 of 2)

SYNC	The clause is ignored.
USAGE	<p>The USAGE options correspond to SQL data types:</p> <ul style="list-style-type: none"> ● COMPUTATIONAL (COMP) or BINARY to SQL type NUMERIC or PICTURE 9's COMP to an integer type (SMALLINT, INTEGER, or LARGEINT). ● DISPLAY to SQL type CHARACTER (for PIC X) or DECIMAL (for PIC 9). ● The INDEX and PACKED-DECIMAL options are not allowed.
VALUE	The clause is ignored.

Using Indicator Variables in a COBOL Program

Null in an SQL column indicates that a value is either unknown or is not applicable. A host language program uses an indicator variable to insert null. It also uses an indicator variable to test for null or a truncated output value.

An indicator variable is an exact numeric variable associated with the host variable that sets or receives the actual column value. The INVOKE directive automatically declares indicator variables for columns that allow null.

A host language program can use an indicator variable to:

- Insert values into a database with an INSERT or UPDATE statement.
- Test for null or a truncated value (in the case of character data) after retrieving a value from a database with a SELECT INTO or FETCH statement.

Inserting Null

To insert values into columns that allow null with an INSERT or UPDATE statement, you must set the indicator variable to a value less than zero for null or zero for a nonnull value before executing the statement.

Examples

A database contains a RETIREES table consisting of the columns EMPNUM and RETIRE_DATE (which allows null). This INSERT statement uses an indicator variable to insert null into the RETIRE_DATE column:

COBOL

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

```
MOVE -1 TO RETIRE-IND.  
EXEC SQL  
    INSERT INTO persnl.retirees  
        VALUES (:NEW-EMPNUM, :RETIRE-DATE INDICATOR :RETIRE-IND)  
END-EXEC.  
...
```

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

```
...  
EXEC SQL  
    INSERT INTO persnl.retirees VALUES (:NEW-EMPNUM, NULL)  
END-EXEC.
```

Testing for Null or a Truncated Value

To test for null or a truncated character value, check the indicator variable associated with a host variable. If the value of the indicator variable is less than zero, the associated column contains null. If the value of the indicator variable is greater than zero, character data in the column was truncated when it was assigned to the host variable.

Example

A database contains a PRODUCTS table that includes the columns PRODNUM and TIMESTAMP_SHIPPED (which allows null). [Example 4-1](#) on page 4-21 selects data from the PRODUCTS table and then tests for null by using the indicator variable SHIP-IND. If the value of the indicator variable is less than 0 (zero), the associated column contains a null value.

COBOL**Example 4-1. Null Test Example**

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 PRODUCT-REC.
    02 PRODNUM          PIC 9(5) COMP.
    02 TIMESTAMP-SHIPPED PIC X(26).
    02 SHIP-IND          PIC S9(4) COMP.
  ...
* Variable for selecting the product number:
  01 MIN-PRODNUM        PIC 9(5) COMP.
  EXEC SQL END DECLARE SECTION END-EXEC.

* Variable for displaying the timestamp or NULL:
  01 VALUE-DISPLAY      PIC X(26) VALUE SPACES.
  ...
* Declare a cursor to perform the SELECT:
  EXEC SQL DECLARE get_prodnum CURSOR FOR
    SELECT prodnum, timestamp_shipped
    FROM sales.products
    WHERE prodnum >= :MIN-PRODNUM
  END-EXEC.

PROCEDURE DIVISION.
0100-MAIN.
  ...
  EXEC SQL OPEN get_prodnum END-EXEC.
  PERFORM 0150-FETCH UNTIL SQLSTATE = "02000".
  EXEC SQL CLOSE get_prodnum END-EXEC.
  ...
0150-FETCH.
  EXEC SQL FETCH get_prodnum INTO
    :PRODNUM,
    DATEFORMAT (:TIMESTAMP-SHIPPED INDICATOR :SHIP-IND, USA)
  END-EXEC.

* SQL/MX sets SHIP-IND to less than zero if the column
* contained a null value in the selected row.
  IF SHIP-IND < 0 THEN MOVE "NULL" TO VALUE-DISPLAY
  ELSE MOVE TIMESTAMP-SHIPPED TO VALUE-DISPLAY.
  IF SQLSTATE = "00000" DISPLAY PRODNUM " " VALUE-DISPLAY.
  ...

```

See DATEFORMAT Function in the *SQL/MX Reference Manual*.

Retrieving Rows With Nulls

To retrieve a row that contains null, use the NULL predicate in the WHERE clause. You cannot use an indicator variable set to -1 in a WHERE clause to retrieve a row that contains null. If you do, NonStop SQL/MX does not find the row and returns a NOTFOUND exception even if a column actually contains null.

Example

To retrieve rows that have null salaries 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:

COBOL

```
* Declare a cursor to find rows with null salaries.
EXEC SQL DECLARE get_null_salary CURSOR FOR
    SELECT empnum, first_name, last_name,
           deptnum, jobcode, salary
    FROM employee
    WHERE salary IS NULL
END-EXEC.

...
PROCEDURE DIVISION.
0100-MAIN.
    ...
    EXEC SQL OPEN get_null_salary END-EXEC.
    PERFORM 200-FETCH-NULL UNTIL SQLSTATE = "02000".
    EXEC SQL CLOSE get_null_salary END-EXEC.
    ...
0200-FETCH-NULL.
    EXEC SQL FETCH get_null_salary INTO
        :EMPNUM OF EMPLOYEE-RECORD,
        :FIRST-NAME OF EMPLOYEE-RECORD
        :LAST-NAME OF EMPLOYEE-RECORD
        :DEPTNUM OF EMPLOYEE-RECORD
        :JOBCODE OF EMPLOYEE-RECORD
        :SALARY OF EMPLOYEE-RECORD
    END-EXEC.
* Process the row that contains the null salary.
...
```

Creating COBOL Host Variables Using INVOKE

The INVOKE preprocessor directive creates host variables corresponding to columns in a table or view. INVOKE converts the column names to COBOL names and generates a COBOL data item for each column. If a column allows null, INVOKE also creates an indicator variable for the column.

You can declare host variables that correspond to the columns in an SQL table or view without using an INVOKE statement. However, using an INVOKE statement to generate host variables has these advantages:

- **Program independence:** If you modify a table or view, the INVOKE statement re-creates the host variables to correspond to the new table or view when you run the SQL/MX COBOL preprocessor. However, you must modify a program that refers to a deleted column or accesses a new column.
- **Performance:** The INVOKE statement maps SQL data types to the corresponding host language data types, and usually no data conversion is required at run time. For further information, see [Example 4-2](#) on page 4-24 and [Example 4-3](#) on page 4-25.

- Program readability and maintenance: The INVOKE statement creates host variables using the same names as column names in the table or view.

Using the INVOKE Directive

To execute an INVOKE directive for a table or view, you must have SELECT privileges to all applicable columns when you run the SQL/MX COBOL preprocessor.

The general syntax for using an embedded INVOKE directive within an SQL Declare Section in a COBOL program is:

```
INVOKE table-or-view AS record
```

For complete syntax, see the INVOKE Directive in the *SQL/MX Reference Manual*.

INVOKE and Date-Time and Interval Host Variables (SQL/MX Release 1.8 Applications)

SQL/MX Release 1.8 does not support SQL:1999 date-time host variables. In SQL/MX Release 1.8 applications, you must declare a character array host variable for date-time or interval data and use the CAST function for input or output from date-time or interval columns.

If your SQL/MX Release 1.8 application uses INVOKE to create a date-time or interval host variable and you plan to preprocess the application in SQL/MX Release 2.x, use the `-e` preprocessor option. Otherwise, SQL/MX Release 2.x returns an error during SQL compilation because the CAST function in the program is incompatible with the SQL:1999 date-time host variables created by INVOKE in SQL/MX Release 2.x.

Note. This issue affects only SQL/MX Release 1.8 applications preprocessed by SQL/MX Release 2.x. Previously compiled SQL/MX Release 1.8 applications continue to run correctly without changes in SQL/MX Release 2.x.

For more information, see [Running the SQL/MX COBOL Preprocessor](#) on page 16-9.

COBOL Record Descriptions Generated by INVOKE

The next examples show the correspondence between a table named SQLCOB_DATA, that contains columns of various SQL data types, and the COBOL record description generated by an INVOKE statement.

This INVOKE statement is coded in the COBOL source file in the SQL Declare Section:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
EXEC SQL INVOKE sqlcob_data AS SQLCOB-TYPES-REC END-EXEC.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The SQL/MX COBOL preprocessor generates the record description immediately after the INVOKE statement in the preprocessed program code.

[Example 4-2](#) shows the CREATE TABLE statement that generates table SQLCOB_DATA.

Example 4-2. CREATE TABLE Statement

```
CREATE TABLE SQLCOB_DATA (
type_char          CHAR (10)                NOT NULL,
type_char_null     CHAR (10)                ,
type_UCS2_CHAR     CHAR(10) CHARACTER SET UCS2 NOT NULL,
type_UCS2_VARCHAR  VARCHAR(10) CHARACTER SET UCS2 NOT NULL,
type_picx          PIC X(10)                NOT NULL,
type_varchar       VARCHAR (10)             NOT NULL,
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_piccomp4_s    PIC S9(2)V9(2)           COMP    NOT NULL,
type_piccomp4_u    PIC 9(2)V9(2)           COMP    NOT NULL,
type_piccomp9_s    PIC S9(7)V9(2)           COMP    NOT NULL,
type_piccomp9_u    PIC 9(7)V9(2)           COMP    NOT NULL,
type_piccomp18_s   PIC S9(16)V9(2)          COMP    NOT NULL,
type_dec4_s        DECIMAL (4)              SIGNED  NOT NULL,
type_dec4_u        DECIMAL (4)              UNSIGNED NOT NULL,
type_dec9_s        DECIMAL (9,2)            SIGNED  NOT NULL,
type_dec9_u        DECIMAL (9,2)            UNSIGNED NOT NULL,
type_dec18_s       DECIMAL (18,2)           SIGNED  NOT NULL,
type_pic4_s        PIC S9(2)V9(2)           NOT NULL,
type_pic4_u        PIC 9(2)V9(2)           NOT NULL,
type_pic9_s        PIC S9(7)V9(2)           NOT NULL,
type_pic9_u        PIC 9(7)V9(2)           NOT NULL,
type_pic18_s       PIC S9(16)V9(2)          NOT NULL,
type_small_s       SMALLINT                 SIGNED  NOT NULL,
type_small_u       SMALLINT                 UNSIGNED NOT NULL,
type_small_null    SMALLINT                 ,
type_int_s         INTEGER                   SIGNED  NOT NULL,
type_int_u         INTEGER                   UNSIGNED NOT NULL,
type_large_s       LARGEINT                 NOT NULL,
type_date          DATE                     NOT NULL,
type_time_6        TIME(6)                 NOT NULL,
type_timestamp_6   TIMESTAMP(6)            NOT NULL,
type_interval      INTERVAL YEAR TO MONTH  NOT NULL
) ;
```

[Example 4-3](#) shows the record generated by this INVOKE directive:

```
EXEC SQL INVOKE sqlcob_data AS SQLCOB-TYPES-REC END-EXEC.
```

Example 4-3. COBOL Record Description Generated by INVOKE (page 1 of 2)

```
* Record Definition for table SQLCOB_DATA
01 SQLCOB-TYPES-REC.
    02 SYSKEY-I          PIC S9(4)          COMP.
    02 SYSKEY            PIC S9(18)         COMP.
    02 TYPE-CHAR         PIC X(10).
    02 TYPE-CHAR-NULL-I  PIC S9(4)          COMP.
    02 TYPE-CHAR-NULL    PIC X(10).
    02 TYPE-UCS2-CHAR     PIC X(20).
    02 TYPE-UCS2-VARCHAR.
        03 LEN           PIC S9(4)          COMP.
        03 VAL           PIC X(20).
    02 TYPE-PICX         PIC X(10).
    02 TYPE-VARCHAR.
        03 LEN           PIC S9(4)          COMP.
        03 VAL           PIC X(10).
    02 TYPE-NUM4-S       PIC S9(4)          COMP.
    02 TYPE-NUM4-U       PIC 9(4)           COMP.
    02 TYPE-NUM9-S       PIC S9(7)V9(2)     COMP.
    02 TYPE-NUM9-U       PIC 9(7)V9(2)     COMP.
    02 TYPE-NUM18-S      PIC S9(16)V9(2)    COMP.
    02 TYPE-PICCOMP4-S   PIC S9(2)V9(2)     COMP.
    02 TYPE-PICCOMP4-U   PIC 9(2)V9(2)     COMP.
    02 TYPE-PICCOMP9-S   PIC S9(7)V9(2)     COMP.
    02 TYPE-PICCOMP9-U   PIC 9(7)V9(2)     COMP.
    02 TYPE-PICCOMP18-S  PIC S9(16)V9(2)    COMP.
    02 TYPE-DEC4-S       PIC S9(4)          DISPLAY SIGN LEADING
                                                SEPARATE.
    02 TYPE-DEC4-U       PIC 9(4)           DISPLAY.
    02 TYPE-DEC9-S       PIC S9(7)V9(2)     DISPLAY SIGN LEADING
                                                SEPARATE.
```

Example 4-3. COBOL Record Description Generated by INVOKE (page 2 of 2)

```

02 TYPE-DEC9-U          PIC 9(7)V9(2)  DISPLAY.
02 TYPE-DEC18-S         PIC S9(16)V9(2) DISPLAY SIGN LEADING
                                SEPARATE.
02 TYPE-PIC4-S          PIC S9(2)V9(2)  DISPLAY SIGN LEADING
                                SEPARATE.
02 TYPE-PIC4-U          PIC 9(2)V9(2)  DISPLAY.
02 TYPE-PIC9-S          PIC S9(7)V9(2)  DISPLAY SIGN LEADING
                                SEPARATE.
02 TYPE-PIC9-U          PIC 9(7)V9(2)  DISPLAY.
02 TYPE-PIC18-S         PIC S9(16)V9(2) DISPLAY SIGN LEADING
                                SEPARATE.
02 TYPE-SMALL-S         PIC S9(4)        COMP.
02 TYPE-SMALL-U         PIC 9(4)        COMP.
02 TYPE-SMALL-NULL-I    PIC S9(4)        COMP.
02 TYPE-SMALL-NULL      PIC S9(4)        COMP.
02 TYPE-INT-S           PIC S9(9)        COMP.
02 TYPE-INT-U           PIC 9(9)        COMP.
02 TYPE-LARGE-S         PIC S9(18)       COMP.
02 TYPE-DATE            PIC X(10).
02 TYPE-TIME-6          PIC X(15).
02 TYPE-TIMESTAMP-6     PIC X(26).
02 TYPE-INTERVAL        PIC X(6).

```

When you use the INVOKE directive to generate host variables, the HP COBOL compiler writes a COBOL data description for each column in the specified table or view. In some cases, the compiler must convert an SQL column name or data type, as described in [Table 4-5](#).

Table 4-5. Changes Made by INVOKE in Generated Host Variables

Column or Data Type	Description of Change
Underscore (_) within a name	Converts underscores to hyphens (-). For example, the column name CITY_STREET becomes CITY-STREET.
Underscore (_) at the end of a name	Truncates the underscore so that the resulting name does not end in a hyphen. For example, the column name HOME_ becomes HOME.
Column with VARCHAR data type	<p>Creates a group item with two elementary data items. The group item name is derived from the VARCHAR column name. The data names of the subordinate data items are:</p> <ul style="list-style-type: none"> ● LEN, a numeric data item for the length ● VAL, a fixed-length character data item for the string, with the maximum length specified by the VARCHAR column definition <p>For example, CUSTNAME defined as VARCHAR (26) becomes this group item:</p> <pre> 01 CUSTNAME . 02 LEN PIC S9(4) COMP . 02 VAL PIC X(26) . </pre>
DATE, TIME, TIMESTAMP, or INTERVAL data type	<p>If the -e preprocessor option is specified, converts columns to character fields.</p> <p>INTERVAL columns have an additional character for a sign. (That is, a negative interval is possible.) The format of the column is the DEFAULT format.</p>

Using Indicator Variables With the INVOKE Directive

The INVOKE directive automatically generates a two-character indicator variable for each host variable that corresponds to a column that allows null. The name of the indicator variable is the same name as the corresponding column, plus a suffix and an optional prefix. If you do not specify a prefix or suffix, the INVOKE statement appends a default suffix (-I for COBOL) to the indicator variable name.

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 the INVOKE statement to generate an indicator variable name derived from the column name and the prefix or suffix. A default suffix of -I applies if the INVOKE directive omits these clauses.

Example

A table named `cob_table` has the columns `empnum` and `empname`. The column `empname` can be null. This example uses an `INVOKE` statement with both the `PREFIX` and `SUFFIX` clauses:

```
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      EXEC SQL INVOKE cob_table PREFIX BEG- SUFFIX -END END-EXEC.
      ...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The SQL COBOL preprocessor generates this record description immediately after the `INVOKE` directive in the preprocessed program code:

```
* Record Definition for table COB_TABLE
01 COB-TABLE.
    02 EMPNUM          PIC 9(4) COMP.
    02 BEG-EMPNAME-END PIC S9(4) COMP.
    02 EMPNAME         PIC X(10).
```

NULL STRUCTURE Clause

The `NULL STRUCTURE` clause causes the `INVOKE` statement to generate a group item for a column that allows null. The group item name is the same as the column name. The group item includes fields for the data item, named `VALU`, and its indicator variable, named `INDICATOR`.

Example

A database contains an `EMPTBL` table consisting of the columns `EMPNUM`, `FIRST_NAME`, `LAST_NAME`, and `HIRE_DATE`. The columns `FIRST_NAME` and `HIRE_DATE` allow null. This example uses an `INVOKE` statement with the `NULL STRUCTURE` clause:

```
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      EXEC SQL INVOKE EMPTBL AS EMPTBL-REC NULL STRUCTURE END-EXEC.
      ...
EXEC SQL END DECLARE SECTION END-EXEC.
```

The SQL/MX COBOL preprocessor generates this record description immediately after the `INVOKE` statement in the preprocessed program code:

```
* Record Definition for table EMPTBL
01 EMPTBL-REC.
    02 EMPNUM          PIC 9(4) COMP.
    02 FIRST-NAME.
        03 INDICATOR   PIC S9(4) COMP.
        03 VALU        PIC X(15).
    02 LAST-NAME       PIC X(20).
    02 HIRE-DATE.
        03 INDICATOR   PIC S9(4) COMP.
        03 VALU        PIC X(10).
```


COBOL Example of Using INVOKE

[Example 4-4](#) on page 4-30 declares and uses host variable names and indicator variable names and shows:

- A host variable declaration with INVOKE that specifies the suffix -I for indicator variables. The invoked record declaration is included as a comment in the example.
- A host variable indicator variable used in the SELECT statement. The columns that might contain null require the indicator variable following the host variable to receive information on nulls.
- An indicator variable testing for null. If the value of the indicator variable following the SELECT is less than zero, the associated column is null.

This record description is similar to the ODETAIL table of the sample database except that the UNIT_PRICE and QTY_ORDERED columns allow null.

Example 4-4. INVOKE Example**COBOL**

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      EXEC SQL
          INVOKE sales.odetail AS ODETAIL-REC SUFFIX -I
      END-EXEC.
* Record Description *****
* 01 ODETAIL-REC.
*    02 ORDERNUM          PIC 9(6) COMP.
*    02 PARTNUM           PIC 9(4) COMP.
*    02 UNIT-PRICE-I      PIC S9(4) COMP.
*    02 UNIT-PRICE        PIC S9(6)V9(2) COMP.
*    02 QTY-ORDERED-I     PIC S9(4) COMP.
*    02 QTY-ORDERED       PIC 9(5) COMP.
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
      ...
* Initialize the host variables in the WHERE clause.
      ...
      EXEC SQL
          SELECT ordernum, partnum, unit_price, qty_ordered
          INTO :ORDERNUM OF ODETAIL-REC,
              :PARTNUM OF ODETAIL-REC,
              :UNIT-PRICE OF ODETAIL-REC
              INDICATOR :UNIT-PRICE-I OF ODETAIL-REC,
              :QTY-ORDERED OF ODETAIL-REC
              INDICATOR :QTY-ORDERED-I OF ODETAIL-REC
          FROM sales.odetail
          WHERE ordernum = :IN-ORDERNUM AND partnum = :IN-PARTNUM
      END-EXEC.
      ...
      IF (UNIT-PRICE-I OF ODETAIL-REC < 0)
          OR (QTY-ORDERED-I OF ODETAIL-REC < 0)
          PERFORM 05000-HANDLE-NULL
      ELSE PERFORM 0300-DISPLAY-RESULT.
      ...

```

Character Set Examples

This set of examples shows methods for selecting, fetching, and inserting data using different character sets. The examples are based on this table:

```

CREATE TABLE STAFF_UC
(EMPNUM   CHAR(3) character set ucs2 NOT NULL UNIQUE,
 EMPNAME  NCHAR VARYING(20),
 GRADE    DECIMAL(4),
 CITY     VARCHAR(15) character set ucs2);

```

Selecting From a UCS2 Character Set Into a VARCHAR Host Variable

This example selects from a UCS2 character set into a VARCHAR host variable with the same length:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 hv-empnum          PIC X(3) CHARACTER SET IS UCS2.
01 hv-empname.
   03 len  PIC S9(4) COMP.
   03 val  PIC X(20) CHARACTER SET UCS2.
01 hv-city.
   03 len  PIC S9(4) COMP.
   03 val  PIC X(15) CHARACTER SET UCS2.
...
EXEC SQL END DECLARE SECTION END-EXEC.

...
* select VARCHAR ucs2 column to VARCHAR UCS2 host variables with
the same length

EXEC SQL
  select empnum,empname,city into :hv-empnum,
    :hv-empname,:hv-city
  from staff_uc
  where empnum < _ucs2'E8' and grade = 10
END-EXEC.

* use the value in hv_empnum, hv_empname, and hv_city
...
```

Fetching From a UCS2 Character Set into a VARCHAR Host Variable

This example shows a FETCH operation from a UCS2 character set into a VARCHAR host variable:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
...
01 hv-empnum          PIC X(3) CHARACTER SET IS UCS2.
01 hv-empname.
   03 len  PIC S9(4) COMP.
   03 val  PIC X(20) CHARACTER SET UCS2.
01 hv-grade PIC S9(4) DISPLAY SIGN LEADING SEPARATE.
01 hv-city.
   03 len  PIC S9(4) COMP.
   03 val  PIC X(15) CHARACTER SET UCS2.
01 hv-cnt  PIC S9(4) COMP.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
.....
* select UCS2 columns into host variable

EXEC SQL
  declare cursor_fetch_ucs2_varchar cursor for
  select * from staff_uc order by empnum
END-EXEC.

EXEC SQL open cursor_fetch_ucs2_varchar END-EXEC.

EXEC SQL fetch cursor_fetch_ucs2_varchar into
  :hv-empnum, :hv-empname, :hv-grade, :hv-city
END-EXEC.

PERFORM UNTIL SQLSTATE NOT = SQLSTATE-OK

* process the output values :hv-empnum, :hv-empname, :hv-grade,
* and :hv-city

  EXEC SQL FETCH cursor_fetch_ucs2_varchar INTO
    :hv-empnum, :hv-empname, :hv-grade, :hv-city
  END-EXEC.
END-PERFORM.
...
```

Simple and Compound Statements

You can access data in an SQL database without a cursor by using simple SQL/MX Data Manipulation Language (DML) statements:

Simple DML Statement

Description

[Single-Row SELECT Statement](#)

Retrieves a single row (or rowset) from a table or view and places the specified column values in host variables. With a cursor, use the DECLARE CURSOR declaration and the FETCH statement.

[INSERT Statement](#)

Inserts one or more rows into a table or view. Use for all INSERT operations.

[Searched UPDATE Statement](#)

Updates the values in one or more columns in a single row or a set of rows of a table or view. With a cursor, use the positioned UPDATE statement.

[Searched DELETE Statement](#)

Deletes a single row or a set of rows from a table or view. With a cursor, use the positioned DELETE statement.

To enable clients to batch multiple SQL statements into one data request to the server, NonStop SQL/MX extends simple DML statements to allow for compound statements, including the assignment statement and the IF statement:

Compound DML Statement

Description

[Compound Statements](#)

Specifies that the BEGIN and END keywords bracket a sequence of SQL statements that must be executed as a single SQL statement. Cannot contain C/C++ or COBOL commands.

[Assignment Statement](#)

In the context of a compound statement, the values computed or set by one SQL statement can be used by subsequent SQL statements within that compound statement.

[IF Statement](#)

In the context of a compound statement, provides conditional execution of SQL statements.

This section describes these two types of statements and also describes PROTOTYPE host variables, which you can use as table names to enable late name resolution for the SQL statements in your program. See [Using PROTOTYPE Host Variables as Table Names](#) on page 5-17.

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 in corresponding host variables. Use this general syntax:

```
SELECT column [,column]...
      INTO :hostvar [,:hostvar]...
      FROM table-name
      WHERE search-condition
```

For complete syntax, see the SELECT statement in the *SQL/MX Reference Manual*.

The search condition is specified so that one row is selected. For information on fetching a set of rows one row at a time by using a SELECT statement that specifies a cursor, see [Section 6, Static SQL Cursors](#).

After a SELECT statement executes, NonStop SQL/MX returns a value to the SQLSTATE variable. If no rows were found satisfying the search condition, the value of SQLSTATE is 02000 (no data). For information on checking the value of SQLSTATE, see [Section 13, Exception Handling and Error Conditions](#).

Using a Primary Key Value to Select Data

You can use a primary key value to select a single row of data.

Example

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

C

```
...
hv_this_employee = input_empnum;          /* set host variable */
...
EXEC SQL SELECT first_name, last_name, deptnum
      INTO :hv_first_name, :hv_last_name, :hv_deptnum
      FROM persnl.employee
      WHERE empnum = :hv_this_employee;
```

Example

Use the SELECT statement to return customer information from the CUSTOMER table by using a primary key value (CUSTNUM column). The WHERE clause specifies that the selected row contains a primary key with a value equal to the host variable named FIND-THIS-CUSTOMER. The SELECT statement retrieves only one row because the primary key value is unique.

COBOL

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

```

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

PROCEDURE DIVISION.
0000-BEGIN.
    EXEC SQL
        WHENEVER NOT FOUND PERFORM 4000-NOT-FOUND
    END-EXEC.
    ...
* Accept input value for host variable in WHERE clause.
    ...
    EXEC SQL
        SELECT custname, street, city, state, postcode
           INTO :CUSTNAME, :STREET, :CITY, :STATE, :POSTCODE
        FROM sales.customer
        WHERE custnum = :FIND-THIS-CUSTOMER
    END-EXEC.
    ...
    DISPLAY CUSTNAME, STREET, CITY, STATE, POSTCODE.
    ...
4000-NOT-FOUND.
    DISPLAY "CUSTOMER NOT FOUND:" FIND-THIS-CUSTOMER.

```

Selecting a Column With Date-Time or INTERVAL Data Type

If a column in the select list has an INTERVAL or standard date-time (DATE, TIME, TIMESTAMP, or the SQL/MP DATETIME equivalents) data type, use the INTERVAL or date-time types. If your C program performs string operations on date-time and INTERVAL host variables, you need to null terminate the date-time and INTERVAL host variables. For information on null termination, see [Fixed-Length Character Data](#) on page 3-17.

If a column in the select list has a nonstandard SQL/MP DATETIME data type that is not equivalent to DATE, TIME, or TIMESTAMP, use the CAST function to convert the column to a character string. You must also specify the length of the target host variable (or the length-1 in the case of a C program) in the AS clause of the CAST conversion.

Standard Date-Time Example

This example uses a typical context for selecting a date-time value:

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
unsigned NUMERIC (4) hv_projcode;
char hv_projdesc[19];
DATE hv_start_date;

```

```
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
  SELECT projcode, projdesc, start_date
  INTO :hv_projcode, :hv_projdesc, :hv_start_date
  FROM samdbcat.persnl.project
  WHERE projcode = 1000;
...
```

Nonstandard SQL/MP DATETIME Example

This example uses a typical context for selecting a nonstandard date-time value, DATETIME MONTH TO DAY:

C

```
EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  unsigned NUMERIC (4) hv_projcode;
  char                hv_projdesc[19];
  char                hv_start_date[6];
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
  SELECT projcode, projdesc, CAST(start_date AS CHAR(5))
  INTO :hv_projcode, :hv_projdesc, :hv_start_date
  FROM samdbcat.persnl.project
  WHERE projcode = 1000;
...
```

Interval Example

This example uses a typical context for selecting an interval value:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 sqlstate                pic x(5).
  01 hv-projcode              pic 9(4) COMP.
  01 hv-projdesc              pic x(18).
  01 hv-est-complete          INTERVAL DAY(4).
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
  EXEC SQL
    SELECT projcode, projdesc, est_complete
    INTO :hv-projcode, :hv-projdesc, :hv-est-complete
    FROM samdbcat.persnl.project
    WHERE projcode = 1000
  END-EXEC.
...
```

INSERT Statement

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

statement to transfer the values from the host variables to the table. Use this general syntax:

```
INSERT INTO table-name (column [,column]...)
VALUES (:hostvar [, :hostvar]...)
```

For complete syntax, see the INSERT statement in the *SQL/MX Reference Manual*.

To execute an INSERT statement, a program must have INSERT privileges for each column in the table receiving the data.

After an INSERT statement executes, NonStop SQL/MX returns a value to the SQLSTATE variable. If a data exception occurs during the insertion process, the value of SQLSTATE is 22xxx (data exception). The class value is 22, and the subclass can be a variety of conditions, depending on the nature of the data being inserted. For information on SQLSTATE values, see [Section 13, Exception Handling and Error Conditions](#).

Inserting Rows

You can insert a single row or multiple rows of data by using the VALUES clause that specifies a row or rows of host variables.

This example inserts a row (the JOBCODE and JOBDESC columns) into the JOB table. The host variables are declared as global host variables, the Declare Section occurs before the definition of main():

Example

C

```
void insert_job(void);
...
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4) hv_jobcode; /* global host variables */
    VARCHAR          hv_jobdesc[19];
EXEC SQL END DECLARE SECTION;
...
int main()
{
    ... /* Input the values of hv_jobcode and hv_jobdesc */
    insert_job();
    ...
    return 0;
} /* end main */

void insert_job(void) {
EXEC SQL INSERT INTO persnl.job (jobcode, jobdesc)
    VALUES (:hv_jobcode, :hv_jobdesc);
} /* end insert_job */
```

Example**COBOL**

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 JOB.
    02 HV-JOBCODE      PIC 9(4) COMP.
    02 HV-JOBDESC     PIC X(18).
  01 SQLSTATE          PIC X(5).
EXEC SQL END DECLARE SECTION END-EXEC.

...
PROCEDURE DIVISION.
  ...
* Move values to HV-JOBCODE and HV-JOBDESC.
  ...
  EXEC SQL INSERT INTO persnl.job (jobcode, jobdesc)
    VALUES (:HV-JOBCODE, :HV-JOBDESC)
  END-EXEC.
  ...

```

Inserting Null

You can insert a row of data with a null column.

Example

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

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
struct emp_tbl {
    unsigned NUMERIC (4)  empnum;
    char                  first_name[16];
    char                  last_name[21];
    ...
} emp;
short ind_1 = -1;
...
EXEC SQL END DECLARE SECTION;
...
/* Enter the values for employee */
printf("\nEnter employee number: ");
scanf("%hu", &emp.empnum);
printf("\nEnter first name: ");
scanf("%s", emp.first_name);
...
blank_pad(emp.first_name, sizeof(emp.first_name) - 1);
blank_pad(emp.last_name, sizeof(emp.last_name) - 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);
...
return 0;
} /* end main */

```

```
void blank_pad(char *buf, size_t size)
{
    size_t i;
    i = strlen(buf);
    if (i < size)
        memset(&buf[i], ' ', size - i);
    buf[size] = '\\0';
} /* end blank_pad */
```

Example

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

C

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

Example

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

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Declare host variables EMPNUM, FIRST-NAME,
* LAST-NAME, DEPTNUM, JOBCODE, and SALARY.
...
01 IND-1   PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
MOVE -1 TO IND-1.
* Move values to host variables EMPNUM, FIRST-NAME,
* LAST-NAME, DEPTNUM, JOBCODE, and SALARY.
...
EXEC SQL INSERT INTO persnl.employee
        VALUES (:EMPNUM, :FIRST-NAME, :LAST-NAME,
                :DEPTNUM, :JOBCODE,
                :SALARY INDICATOR :IND-1)

END-EXEC.
```

Example

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

COBOL

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

END-EXEC.
```

Inserting a Date-Time Value

For standard date-time columns (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents), you insert a row directly with the date-time host variable. For

nonstandard SQL/MP DATETIME columns that are not equivalent to DATE, TIME, or TIMESTAMP, use the CAST function to insert a row with a date-time value.

Standard Date-Time Example

This example inserts a new row into the PROJECT table, including a timestamp value in the SHIP_TIMESTAMP column:

```
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      01 HV-TIMESTAMP    TIMESTAMP.
      ...
EXEC SQL END DECLARE SECTION END-EXEC.
      ...
* Initialize host variables for new row
      ...
      MOVE "1997-04-25 08:14:12.000000" TO HV-TIMESTAMP.
      ...
      EXEC SQL INSERT INTO PROJECT
        (... , SHIP_TIMESTAMP, ...)
        VALUES(... , :HV-TIMESTAMP), ...)
      END-EXEC.
      ...
```

Nonstandard SQL/MP DATETIME Example

This example inserts a new row into the PROJECT table, including a nonstandard time value, DATETIME HOUR TO SECOND, in the SHIP_TIMESTAMP column:

```
C EXEC SQL BEGIN DECLARE SECTION;
   char hv_timestamp[9];
   ...
EXEC SQL END DECLARE SECTION;
   ...
   /* Initialize host variables for new row */
   strcpy(hv_timestamp, "1997-04-25 08:14:12.000000");
   ...
EXEC SQL INSERT INTO PROJECT
  (... , SHIP_TIMESTAMP, ...)
  VALUES(... , CAST(:hv_timestamp AS DATETIME HOUR TO SECOND),
    ...);
   ...
```

Inserting an Interval Value

Insert a row directly with the INTERVAL host variable.

C Interval Example

A table includes a column with a year-month interval and a column with a day-time interval. This example inserts a new row into this table:

```
C EXEC SQL BEGIN DECLARE SECTION;
   INTERVAL YEAR TO MONTH    hv_year_month;
```

```

        INTERVAL DAY TO SECOND(4)  hv_day_time;
    ...
EXEC SQL END DECLARE SECTION;
...
    /* Initialize host variables for new row */
strcpy(hv_year_month, "63-04");
strcpy(hv_day_time, "25:08:14:12.0000");
...
EXEC SQL INSERT INTO RETIREES
    (... , AGE, LAST_TIMECARD)
    VALUES(... , :hv_year_month,
              :hv_day_time);
...

```

COBOL Interval Example

A table includes a column with a year-month interval and a column with a day-time interval. This example inserts a new row into this table:

```

COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01 HV-YEAR-MONTH   INTERVAL DAY TO MONTH.
        01 HV-DAY-TIME     INTERVAL DAY TO SECOND(4).
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
* Initialize host variables for new row
...
MOVE "63-04" TO HV-YEAR-MONTH.
MOVE "25:08:14:12.0000" TO HV-DAY-TIME.
...
EXEC SQL INSERT INTO RETIREES
    (... , AGE, LAST_TIMECARD)
    VALUES(... ,
              :HV-YEAR-MONTH,
              :HV-DAY-TIME)
END-EXEC.
...

```

Searched UPDATE Statement

The searched UPDATE statement updates the values in one or more columns in either a single row or in a set of rows of a table. The selection of the rows to be updated is based on a search condition. Use this general syntax:

```

UPDATE table-name
SET set-clause-list WHERE search-condition

```

For complete syntax, see UPDATE statement in the *SQL/MX Reference Manual*. To update a set of rows one row at a time by using a cursor, see [Section 6, Static SQL Cursors](#).

To execute an UPDATE statement, a program must have UPDATE privileges on each column being updated in the table.

After an UPDATE statement executes, NonStop SQL/MX returns a value to the SQLSTATE variable. If no rows were found satisfying the search condition, the value of SQLSTATE is 02000 (no data). If a data exception occurs during the update process, the value of SQLSTATE is 22xxx. The class value is 22, and the subclass can be a variety of conditions, depending on the nature of the data being inserted. For information on SQLSTATE values, see [Table 13-1](#) on page 13-2.

Updating a Single Row

You can update a single row of data.

This example updates a single row of the ORDERS table that contains information on the order number specified by update_ordernum:

C Searched UPDATE Example

C

```
void update_orders(void);

EXEC SQL BEGIN DECLARE SECTION;
DATE          update_date;
unsigned long  update_ordernum;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

...
int main()
{
...
/* Input the values of update_date and update_ordernum */
update_orders();
...
return 0;
} /* end main */

void update_orders(void) {
EXEC SQL UPDATE sales.orders
SET deliv_date = :update_date
WHERE ordernum = :update_ordernum
FOR READ COMMITTED ACCESS;
} /* end update_orders */
```

COBOL Searched UPDATE Example

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE          PIC X(5).
01 UPDATE-DATE        DATE.
01 UPDATE-ORDERNUM    PIC 9(6) COMP.
```

```
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
...
EXEC SQL UPDATE sales.orders
  SET deliv-date = :UPDATE-DATE
  WHERE ordernum = :UPDATE-ORDERNUM
  FOR READ COMMITTED ACCESS
END-EXEC.
```

Updating Multiple Rows

If you do not need to check a value in a row before you update the row, use a single UPDATE statement to update multiple rows in a table.

Examples

This example updates the SALARY column of all rows in the EMPLOYEE table where the SALARY value is less than the hv_min_salary host variable. A user enters the values for hv_inc and hv_min_salary:

```
C EXEC SQL UPDATE persnl.employee
      SET salary = salary * :hv_inc
      WHERE salary < :hv_min_salary;
```

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

```
COBOL EXEC SQL UPDATE persnl.employee
      SET deptnum = :HV-NEW-DEPTNUM
      WHERE deptnum = :HV-OLD-DEPTNUM
END-EXEC.
```

Updating Columns To Null

You can update a value in a row of data to null.

Examples

This example updates the specified SALARY column to null by using an indicator variable. The set_to_null host variable specifies the row to update:

```
C /* ind_var is set to -1 */
EXEC SQL UPDATE persnl.employee
  SET salary = :emp_tbl.salary INDICATOR :ind_var
  WHERE jobcode = :set_to_null;
```

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

```
COBOL EXEC SQL UPDATE persnl.employee
      SET salary = NULL
```

```
WHERE jobcode = :SET-TO-NULL
END-EXEC.
```

Searched DELETE Statement

The searched DELETE statement deletes one or more rows from a table. The selection of the rows to be deleted is based on a search condition. If you delete all rows from a table, the table still exists until it is deleted by a DROP TABLE statement. Use this general syntax:

```
DELETE FROM table-name
WHERE search-condition
```

For complete syntax, see the DELETE statement in the *SQL/MX Reference Manual*.

To delete a set of rows one row at a time by using a cursor, see [Section 6, Static SQL Cursors](#).

To execute a DELETE statement, a program must have DELETE privileges on the table.

After a DELETE statement executes, NonStop SQL/MX returns a value to SQLSTATE. If no rows were found satisfying the search condition, the value of SQLSTATE is 02000 (no data). For information on SQLSTATE values, see [Table 13-1](#) on page 13-2.

Deleting a Single Row

To delete a single row, move a unique key value to a host variable and then specify the host variable in the WHERE clause.

Example

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 hv_empnum host variable:

C

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

Deleting Multiple Rows

If you do not need to check a column value before you delete a row, you can use a single DELETE statement to delete multiple rows in a table.

Examples

This example deletes all rows (or employees) from the EMPLOYEE table specified by the `deptnum_to_delete` host variable (which is entered by a user):

C

```
EXEC SQL
  DELETE FROM persnl.employee
  WHERE deptnum = :deptnum_to_delete;
```

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

COBOL

```
EXEC SQL
  DELETE FROM invent.partsupp
  WHERE partnum BETWEEN :TERMINAL-FIRST-NUM
                     AND :TERMINAL-LAST-NUM
                     AND partcost > :TERMINAL-MAX-COST

END-EXEC.
```

Compound Statements

Compound statements enable SQL/MX clients to batch multiple SQL statements into one data request to the server. This reduction in the number of client-server requests results in increased transaction throughput and, consequently, faster response times. The compound statement feature benefits both the client and the server. The client waits for the server a reduced number of times, and the server switches contexts a reduced number of times when responding to different client requests.

Compound statements are supported for static SQL only.

The SQL statements are coded within the `BEGIN` and `END` keywords. The statements are executed sequentially and are atomic. Therefore, if the execution of any statement within the `BEGIN` and `END` keywords encounters an error (such as a unique constraint violation), NonStop SQL/MX automatically rolls back all the statements. NonStop SQL/MX limits the quantity of SQL statements in a compound statement to 100.

Each `SELECT` statement within a `BEGIN ... END` statement should return at least one row. If a `SELECT` statement within a `BEGIN ... END` statement does not return at least one row, further execution of the compound statement stops, and NonStop SQL/MX issues either a warning (8014) or an error (8015). The warning is displayed if no updates occurred before the `SELECT` statement that did not return a row. In the case of the warning, NonStop SQL/MX does not roll back the transaction. The error is displayed if updates occurred before the `SELECT` statement that did not return a row. Because the updates occurred as part of this compound statement, NonStop SQL/MX rolls back the transaction. In both cases, the behavior is atomic because none of the statements are executed.

```
BEGIN
  SQL-statement ; [SQL-statement ; ] ...
END;
```

For complete syntax, see the Compound (BEGIN ... END) statement in the *SQL/MX Reference Manual*.

Although you cannot use cursors in compound statements, you can use rowsets to retrieve multiple rows from database tables. You cannot embed C/C++ or COBOL commands within a compound statement.

You cannot use a compound statement within trigger actions. The INSERT, UPDATE, and DELETE statements cannot be trigger events when they are used in a compound statement.

Example

Group three statements—two INSERT statements and an UPDATE statement—that update the database within a single transaction. If an error occurs within the compound statement, program control continues following the compound statement, and the application issues a rollback to undo the effects of the other statements:

C

```
EXEC SQL WHENEVER SQLERROR GOTO end_compound;

EXEC SQL BEGIN WORK;

EXEC SQL BEGIN
    INSERT INTO sales.orders VALUES (:ordernum, DATE '1998-03-23',
        DATE '1998-03-30', 75, 7654);

    INSERT INTO sales.odetail VALUES (:ordernum, :partnum,
        :price, :qty);

    UPDATE invent.partloc SET QTY_ON_HAND = QTY_ON_HAND - :qty
        WHERE PARTNUM = :partnum AND LOC_CODE = 'G45';
END;

end_compound:
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
    EXEC SQL COMMIT WORK;          /* Commit the changes */
else
    EXEC SQL ROLLBACK WORK;        /* Roll back the changes */
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE      PIC X(5).
01 SQLSTATE-OK   PIC X(5) VALUE "00000".
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
01-start-compound.
EXEC SQL WHENEVER SQLERROR GO TO 09-end-compound END-EXEC.
EXEC SQL BEGIN WORK END-EXEC.
EXEC SQL BEGIN
    INSERT INTO sales.orders VALUES (:ordernum, DATE '1998-03-23',
        DATE '1998-03-30', 75, 7654);
    INSERT INTO sales.odetail VALUES (:ordernum, :partnum,
        :price, :qty);
    UPDATE invent.partloc SET QTY_ON_HAND = QTY_ON_HAND - :qty
        WHERE PARTNUM = :partnum AND LOC_CODE = 'G45';
END END-EXEC.
```

```

09-end-compound.
    IF SQLSTATE = SQLSTATE-OK
*       Commit the change
        EXEC SQL COMMIT WORK END-EXEC
    ELSE
*       Roll back the change
        EXEC SQL ROLLBACK WORK END-EXEC.

```

NonStop SQL/MX supports the use of host variables but does not allow for local variables declared within the BEGIN and END keywords.

You can use SELECT statements inside compound statements only if each SELECT retrieves at most one row (or rowset) result. This restriction is the normal requirement for using the SELECT INTO statement.

Example

Use the SELECT INTO statement to retrieve order information from the database and then update the quantity on hand for that part number:

C

```

EXEC SQL BEGIN;
    SELECT UNIT_PRICE, QTY_ORDERED
        INTO :unit_price, :qty_ordered
        FROM ODETAIL
        WHERE ORDERNUM = :ordernum AND PARTNUM = :partnum;

    UPDATE PARTLOC
        SET QTY_ON_HAND = QTY_ON_HAND - :qty_ordered
        WHERE PARTNUM = :partnum AND LOC_CODE = :loc_code;
END;

```

COBOL

```

EXEC SQL BEGIN
    SELECT UNIT_PRICE, QTY_ORDERED
        INTO :unit-price, :qty-ordered
        FROM ODETAIL
        WHERE ORDERNUM = :ordernum AND PARTNUM = :partnum;
    UPDATE PARTLOC
        SET QTY_ON_HAND = QTY_ON_HAND - :qty-ordered
        WHERE PARTNUM = :partnum AND LOC_CODE = :loc-code;
END END-EXEC.

```

Assignment Statement

Inside a compound statement, an SQL statement can compute and assign the value of an expression to a host variable. Subsequent SQL statements inside that compound statement can then use that host variable to get the value of the expression computed by the preceding SQL statement. Use this general syntax:

<pre>SET <i>assignment-target</i> = <i>assignment-source</i></pre>
--

The target side of the assignment is a list of host variables. The source side of the assignment is a value expression, NULL, or a row subquery.

For complete syntax, see the Assignment statement in the *SQL/MX Reference Manual*.

Example

This example retrieves order information from the database and then updates the quantity on hand for that part number. You can use the SET statement as an alternative to the SELECT INTO statement:

C

```
EXEC SQL BEGIN;
  SET :unit_price,:qty_ordered = SELECT UNIT_PRICE, QTY_ORDERED
    FROM ODETAIL
    WHERE ORDERNUM = :ordernum AND PARTNUM = :partnum;

  UPDATE PARTLOC
    SET QTY_ON_HAND = QTY_ON_HAND - :qty_ordered
    WHERE PARTNUM = :partnum AND LOC_CODE = :loc_code;
END;
```

COBOL

```
EXEC SQL BEGIN
  SET :unit-price, :qty-ordered = SELECT UNIT_PRICE, QTY_ORDERED
    FROM ODETAIL
    WHERE ORDERNUM = :ordernum AND PARTNUM = :partnum;

  UPDATE PARTLOC
    SET QTY_ON_HAND = QTY_ON_HAND - :qty-ordered
    WHERE PARTNUM = :partnum AND LOC_CODE = :loc-code;
END END-EXEC.
```

IF Statement

An IF statement provides branching inside compound statements. An IF statement is a compound statement that provides conditional execution based on the truth value of a conditional expression. Use this general syntax:

```
IF conditional-expression THEN
  SQL-statement;[SQL-statement;]...
  [ELSEIF conditional-expression THEN
    SQL-statement;[SQL-statement;]......]...
  [ELSE SQL-statement;[SQL-statement;]...]
END IF
```

For the complete syntax and semantics, see the IF statement in the *SQL/MX Reference Manual*.

Example

In this example, INSERT and SELECT statements execute sequentially only for new orders. Otherwise, the SELECT statement returns information on the current customer:

C

```
...
EXEC SQL
BEGIN
IF :hv_new_ordernum <> 0
THEN
  INSERT INTO SALES.ORDERS
```

```

        (ORDERNUM, ORDER_DATE, DELIV_DATE, SALESREP, CUSTNUM)
        VALUES (:hv_new_ordernum, :hv_orderdate, :hv_delivdate,
                :hv_salesrep, :hv_custnum);
SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
    INTO :hv_custnum, :hv_custname,
        :hv_street, :hv_city, :hv_state, :hv_postcode
    FROM SALES.CUSTOMER
    WHERE CUSTNUM = :hv_custnum;
ELSE
SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
    INTO :hv_custnum, :hv_custname,
        :hv_street, :hv_city, :hv_state, :hv_postcode
    FROM SALES.CUSTOMER
    WHERE CUSTNUM = :hv_current_custnum;

END IF;
END;
...

```

COBOL

```

EXEC SQL
BEGIN
IF :hv-new-ordernum <> 0
THEN
    INSERT INTO SALES.ORDERS
    (ORDERNUM, ORDER_DATE, DELIV_DATE, SALESREP, CUSTNUM)
    VALUES (:hv-new-ordernum, :hv-orderdate, :hv-delivdate,
            :hv-salesrep, :hv-custnum);
SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
    INTO :hv-custnum, :hv-custname,
        :hv-street, :hv-city, :hv-state, :hv-postcode
    FROM SALES.CUSTOMER
    WHERE CUSTNUM = :hv-custnum;
ELSE
SELECT CUSTNUM, CUSTNAME, STREET, CITY, STATE, POSTCODE
    INTO :hv-custnum, :hv-custname,
        :hv-street, :hv-city, :hv-state, :hv-postcode
    FROM SALES.CUSTOMER
    WHERE CUSTNUM = :hv-current-custnum;
END IF;
END END-EXEC.

```

Using PROTOTYPE Host Variables as Table Names

You can dynamically change the name of a table or view in an embedded SQL statement by using a host variable to provide the table name during execution. This capability enables late name resolution. For more information, see [Late Name Resolution](#) on page 8-6.

After you declare a host variable for the table name, you can specify it within an embedded SQL statement by using the PROTOTYPE clause. For the syntax, see [PROTOTYPE Host Variables For SQL/MP and SQL/MX Objects](#) on page 8-4.

You must initialize the value of the PROTOTYPE host variable before the execution of the embedded SQL statement.

Example

This example selects like columns from multiple tables that are specified dynamically. There is a separate job code table for each division within a corporation:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR          hv_tablename[161];
    unsigned NUMERIC (4) hv_jobcode;
    VARCHAR          hv_jobdesc[19];
    ...
EXEC SQL END DECLARE SECTION;

...
/* Initialize prototyped host variable name for the table. */
printf("Enter the fully qualified name of the table: ");
scanf("%s", &hv_tablename);

/* Initialize host variable in WHERE clause. */
printf("Enter job code to be retrieved: ");
scanf("%hu", &hv_this_jobcode);
...
EXEC SQL
    SELECT jobcode, jobdesc
    INTO :hv_jobcode, hv_jobdesc
    FROM :hv_tablename PROTOTYPE 'samdbcat.persnl.job'
    WHERE jobcode = :hv_this_jobcode;
...
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 hv-tablename      PIC X(160).
01 hv-jobcode        PIC 9(4).
01 hv-this-jobcode   PIC 9(4).
01 hv-jobdesc.
    02 LEN           PIC S9(4) COMP.
    02 VAL           PIC X(18).
...
EXEC SQL END DECLARE SECTION END-EXEC.

...
* Initialize prototyped host variable name for the table.

MOVE ALL SPACES TO hv-tablename.
DISPLAY "Enter the fully qualified name of the table: ".
ACCEPT hv-tablename.

* Initialize host variable in WHERE clause.

DISPLAY "Enter job code to be retrieved: ".
```

```
ACCEPT hv-this-jobcode.  
...  
EXEC SQL  
  SELECT jobcode, jobdesc  
  INTO :hv-jobcode, :hv-jobdesc  
  FROM :hv-tablename PROTOTYPE 'samdbcat.persnl.job'  
  WHERE jobcode = :hv-this-jobcode END-EXEC.
```

The columns that you select from a dynamically specified table must be the same as the columns in the table specified in the PROTOTYPE clause during static compilation. Otherwise, NonStop SQL/MX returns an error.

6

Static SQL Cursors

In NonStop SQL/MX, a mechanism called a cursor allows an application program to select and then retrieve a set of rows one row at a time. Each row in the set satisfies the criteria in the search condition of the SELECT statement that specifies the cursor. NonStop SQL/MX builds a result table to hold all the rows retrieved by executing the SELECT statement and then uses a cursor to make rows from the result table available to your program. The cursor identifies the current row of the result table.

The SELECT statement that specifies the cursor must be within a DECLARE CURSOR declaration, which defines and names the cursor, identifying the set of rows to be retrieved.

The result table of a cursor is processed like a sequential data set. First open the cursor with an OPEN statement before any rows are retrieved. Then use a FETCH statement to retrieve the cursor's current row. The program can test the data in each row at the current cursor position and then, if the data meets certain criteria, it can display, update, delete, or bypass the row. Use FETCH repeatedly until all rows have been retrieved. When you have finished processing the rows, close the cursor with a CLOSE statement.

This section describes:

- [DML Statements for Static SQL Cursors](#) on page 6-1
- [Steps for Using a Static SQL Cursor](#) on page 6-2
- [Using Date-Time and INTERVAL Data Types](#) on page 6-12
- [Considerations When Using a Cursor](#) on page 6-14

DML Statements for Static SQL Cursors

The DML statements you can use with static SQL cursors are:

DECLARE CURSOR	Defines a cursor and associates it with a query expression.
OPEN	Opens a cursor.
FETCH	Positions a cursor on the next row of a table and retrieves values from the row.
Positioned UPDATE	Updates a row from a table or view at the current cursor position.
Positioned DELETE	Deletes a row from a table or view at the current cursor position.
CLOSE	Closes a cursor.

For detailed statement descriptions, see the *SQL/MX Reference Manual*.

Note. Using a cursor can sometimes degrade performance. A cursor requires OPEN, FETCH, and CLOSE statements, which increase the number of messages between the HP NonStop Distribution Service (DS) and the HP NonStop Data Access Manager (DAM). Consider not using a cursor if a single-row retrieval is sufficient.

Steps for Using a Static SQL Cursor

[Figure 6-1](#) shows the steps presented within the complete C program. These steps are executed in the sample program [Example A-1](#) on page A-1.

c

Figure 6-1. Using a Static SQL Cursor in a C Program

```

1  ...
   EXEC SQL BEGIN DECLARE SECTION;
   int hostvar;
   ...
   EXEC SQL END DECLARE SECTION;
   ...

2  EXEC SQL DECLARE sql_cursor CURSOR FOR
      SELECT column_1, column_2, ..., column_n
      FROM catalog.schema.table
      WHERE column_1 = :hostvar;

   ...
   void find_row(void)
   {
   ...

3  hostvar = initial_value;
   ...

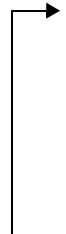
4  EXEC SQL OPEN sql_cursor;
   ...

5  EXEC SQL FETCH sql_cursor
      INTO :hostvar_1, :hostvar_2, ..., :hostvar_n;
   ...
6  ... /* Process values in the host variable(s). */
   ...
   ... /* If last row has not been processed, */
   ... /* branch back to fetch another row. */
7  ...

8  EXEC SQL CLOSE sql_cursor;
   ...
   } /* end find_row
   ...

```

[Figure 6-2](#) shows the steps presented within the complete COBOL program. These steps are executed in the sample program [Example C-1](#) on page C-1.

COBOL**Figure 6-2. Using a Static SQL Cursor in a COBOL Program**


```

1      ...
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 HOSTVAR  9(4) COMP.
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
      ...

2      EXEC SQL DECLARE sql_cursor CURSOR FOR
          SELECT column_1, column_2, ..., column_n
          FROM catalog.schema.table
          WHERE column_1 = :HOSTVAR
      END-EXEC.
      ...

3      MOVE INITIAL-VALUE TO HOSTVAR.
      ...

4      EXEC SQL OPEN sql_cursor END-EXEC.
      ...

5      EXEC SQL FETCH sql_cursor
          INTO :HOSTVAR-1, :HOSTVAR-2, ..., :HOSTVAR-N
      END-EXEC.
      ...
      * Process values in the host variable(s).
6      ...
      * If last row has not been processed,
      * branch back to fetch another row.
7      ...

8      EXEC SQL CLOSE sql_cursor END-EXEC.
      ...
  
```

For more information:

1. [Declare Required Host Variables](#) on page 6-4
2. [Declare the Cursor](#) on page 6-4
3. [Initialize the Host Variables](#) on page 6-5
4. [Open the Cursor](#) on page 6-5
5. [Retrieve the Values](#) on page 6-6
6. [Process the Retrieved Values](#) on page 6-7
7. [Fetch the Next Row](#) on page 6-10
8. [Close the Cursor](#) on page 6-11

Declare Required Host Variables

In an SQL Declare Section, declare any host variables you specify in the query expression of the DECLARE CURSOR declaration:

- Before the DECLARE CURSOR declaration
- Within the same scope as the SQL statements that refer to them

Declare the Cursor

Use the DECLARE CURSOR declaration to name a cursor and associate it with a query expression. You can specify a row order for the result table of the query expression. You can also specify the result table as read-only or to enable the update of specific columns. Use this general syntax:

```
DECLARE cursor-name CURSOR FOR query-expression
  [ORDER BY sort-specification]
  [FOR {READ ONLY | UPDATE [OF column-name-list]}]
```

For complete syntax, see the DECLARE CURSOR Declaration in the *SQL/MX Reference Manual*.

The name of a static cursor is an SQL identifier. The query expression typically consists of a SELECT statement that specifies the rows that a subsequent FETCH statement retrieves, one row at a time. The FETCH statement can also store the retrieved table values into host variables, which your host language program can then process. For example, your program can list, update, delete, or save the values in an array.

You code a DECLARE CURSOR declaration:

- In listing order before other SQL statements that refer to the cursor, including the OPEN, FETCH, DELETE, UPDATE, and CLOSE statements
- Within the scope of other SQL statements that refer to the cursor

Example

This example declares a read-only cursor named `get_name_address` that accesses the CUSTOMER table. The query expression specifies the name and address of all customers within a certain range of zip codes. The BETWEEN clause specifies the range, and the ORDER BY clause sorts the rows by zip code:

C

```
EXEC SQL BEGIN DECLARE SECTION;
char first_postcode[11], last_postcode[11];
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE get_name_address CURSOR FOR
  SELECT custname,street,city,state,postcode
  FROM customer
  WHERE postcode BETWEEN :first_postcode AND :last_postcode
```

```
ORDER BY postcode
FOR READ ONLY;
```

Example

This example declares an updatable cursor named `get_by_partnum` that accesses the PARTS table. The query expression specifies all part numbers greater than or equal to the host variable named `min-partnum`:

```
C EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 min-partnum                pic 9(4) COMP.
    ...
EXEC SQL END DECLARE SECTION END-EXEC.
    ...
    EXEC SQL DECLARE get_by_partnum CURSOR FOR
        SELECT partnum, partdesc, price, qty_available
        FROM   parts
        WHERE  partnum >= :min-partnum
        FOR UPDATE OF price, qty_available
    END-EXEC.
    ...
```

Initialize the Host Variables

Initialize the host variables you specified in the query expression in the DECLARE CURSOR declaration. You must initialize the host variables before you execute the OPEN statement, or 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. As a result, the FETCH might not begin at the expected location in the result table.

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

Open the Cursor

Use the OPEN statement to establish the result table and position the cursor before the first row of the table. You can sort the result table if the query expression specified in the cursor declaration includes the ORDER BY clause. Use this general syntax:

```
OPEN cursor-name
```

For complete syntax, see the OPEN statement in the *SQL/MX Reference Manual*.

For audited tables or views, use the OPEN statement to associate a cursor with a TMF transaction. SQL/MX format tables and views are always audited. SQL/MP format tables provide a choice of whether to audit.

The OPEN statement must execute before any FETCH statements for the cursor and within the scope of all other SQL statements that refer to the cursor, including DECLARE CURSOR, FETCH, UPDATE, DELETE, and CLOSE statements.

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.)

Example

This example opens the `get_name_address` cursor:

```
EXEC SQL OPEN get_name_address;
```

Retrieve the Values

Use the FETCH statement to position the cursor at the next row of the result table and to transfer the values defined in the query expression of the cursor declaration to the corresponding host variables. Use this general syntax:

```
FETCH cursor-name INTO :hostvar [,:hostvar ]...
```

For complete syntax, see the OPEN statement in the *SQL/MX Reference Manual*.

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

For audited tables or views, the FETCH statement must execute within the same transaction as the OPEN statement. This is not true for WITH HOLD cursors. After the FETCH statement has retrieved all rows specified by the query expression, a subsequent FETCH causes a no-data exception (SQLSTATE equal to 02000).

This example retrieves information from the PARTS table:

Example

```
C EXEC SQL BEGIN DECLARE SECTION;
struct parts_type {           /* host variables */
    unsigned short partnum;
    char            partdesc[19];
    long            price;
    long            qty_available
} ;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE list_by_partnum CURSOR FOR
    SELECT partnum, partdesc, price, qty_available
    FROM   parts
    WHERE  partnum >= :parts_rec1.partnum
    ORDER BY partnum
    READ ONLY;
```

```

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

```

Example

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 parts-recl.
    02 hv-partnum          pic 9(4) COMP.
    02 hv-partdesc        pic x(18).
    02 hv-price           pic s9(6)v9(2) COMP.
    02 hv-qty-available   pic s9(5) COMP.
  01 parts-rec2.
    ...
  01 min-partnum          pic 9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
...
PROCEDURE DIVISION.
  ...
  EXEC SQL DECLARE list_by_partnum CURSOR FOR
    SELECT partnum, partdesc, price, qty_available
    FROM   parts
    WHERE  partnum >= :min-partnum
    ORDER BY partnum
    FOR READ ONLY
  END-EXEC.
  ...
  * Set value for min-partnum.
  ...
  EXEC SQL OPEN list_by_partnum END-EXEC.
  EXEC SQL
    FETCH list_by_partnum
    INTO :hv-partnum OF parts-recl,
        :hv-partdesc OF parts-recl,
        :hv-price OF parts-recl,
        :hv-qty-available OF parts-recl
  END-EXEC.
  ...

```

Process the Retrieved Values

After the FETCH statement returns the values to the host variables, your program can process the values. For example, you can test one or more values and then perform one of these operations:

- Update columns in the current row by using a positioned UPDATE statement.
- Delete the current row by using a positioned DELETE statement.
- List or display the values.

- Save the values in an array and process them later.

After you process a row, retrieve the next row by using the `FETCH` statement. Continue executing this loop until you have processed all rows specified by the query expression. After all rows have been processed, `SQLSTATE` is `02000`, and `SQLCODE` is 100.

Positioned UPDATE Statement

Use the positioned `UPDATE` statement to update a row in a table at the current cursor position. You can update multiple rows, one row at a time. Before you update a row, you can also test one or more column values if necessary. Use this general syntax:

```
UPDATE table-name
  SET column = :hostvar [,column = :hostvar ]...
 WHERE CURRENT OF cursor-name
```

For complete syntax, see the `UPDATE` statement in the *SQL/MX Reference Manual*.

The `WHERE CURRENT OF` clause specifies the row to update. The `SET` clause updates each column in the current row by using the new values in the host variables. You can use the `WHERE CURRENT OF` syntax only with simple `SELECT` statements.

To execute an `UPDATE` statement, the cursor must be declared as updatable (the default is read-only) in the query expression of the `DECLARE CURSOR` declaration.

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

For audited tables and views, the `UPDATE` statement must also execute within the same transaction as the `OPEN` and `FETCH` statements for the cursor.

After the positioned `UPDATE` statement executes, the cursor remains positioned on the current row.

Example

Use the `get_by_partnum` cursor and the host variables named `new_price` and `new_qty` to update the `PARTS` table:

C

```
EXEC SQL DECLARE get_by_partnum CURSOR FOR
      SELECT partnum, partdesc, price, qty_available
      FROM sales.parts
      WHERE partnum >= :min_partnum
      FOR UPDATE OF price, qty_available;
... /* Set the value of min_partnum host variable */
EXEC SQL OPEN get_by_partnum;
EXEC SQL FETCH get_by_partnum INTO ... ;
... /* Test value(s) in the current row. */
... /* Set new values in the host variables. */
/* Update the current row. */
EXEC SQL UPDATE sales.parts
```



```

        SET price          = :new_price,
           qty_available = :new_qty
        WHERE CURRENT OF get_by_partnum;

... /* Branch back to retrieve the next row. */

EXEC SQL CLOSE get_by_partnum;

```

Positioned DELETE Statement

Use the positioned DELETE statement to delete a row in a table at the current cursor position. You can delete multiple rows, one row at a time. Before you delete a row, you can also test one or more column values if necessary. Use this general syntax:

```

DELETE FROM table-name
WHERE CURRENT OF cursor-name

```

For complete syntax, see the DELETE statement in the *SQL/MX Reference Manual*.

The WHERE CURRENT OF clause specifies the row to delete. The cursor must also be declared as updatable (the default is read-only) in the query expression of the DECLARE CURSOR statement.

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

A positioned DELETE statement must execute within the scope of other SQL statements that refer to the cursor, including the DECLARE CURSOR, OPEN, and FETCH statements.

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

After the positioned DELETE statement executes, the cursor is positioned before the next row in the result table.

Example

This example declares a cursor named `get_by_partnum`, retrieves data from the PARTS table, tests the data, and deletes specific rows:

```
PROCEDURE DIVISION.
```

COBOL

```

        ...
EXEC SQL DECLARE get_by_partnum CURSOR FOR
        SELECT partnum, partdesc, price, qty_available
        FROM sales.parts
        WHERE partnum >= :min-partnum

END-EXEC.

        ...
EXEC SQL OPEN get_by_partnum END-EXEC.
EXEC SQL FETCH get_by_partnum ... END-EXEC.
* Test the value(s) in the current row.
        ...
* Delete the current row.

```

```

EXEC SQL DELETE FROM sales.parts
      WHERE CURRENT OF get_by_partnum
END-EXEC.
* Branch back to retrieve the next row.
...
EXEC SQL CLOSE get_by_partnum END-EXEC.

```

Fetch the Next Row

In [Retrieve the Values](#) on page 6-6, the `FETCH` statement positions the cursor at the next row of the result table and transfers the values defined in the query expression of the `DECLARE CURSOR` statement to the corresponding host variables. After the `FETCH` statement has retrieved all rows specified by the query expression, a subsequent `FETCH` causes a no-data exception (SQLSTATE equal to 02000).

This example uses SQLSTATE to control a `while` loop. Fetch the first row of the result table of the query expression. Following this first `FETCH` statement, construct a `while` loop that executes provided SQLSTATE returns the 00000 value.

Within the `while` loop, your program processes the fetched row and then executes another fetch for the next row of the result table. Following the `while` loop (when SQLSTATE is no longer 00000), you can test for the end no-data condition for the cursor and branch to the `CLOSE` statement if no more data is available:

Examples

C

```

EXEC SQL BEGIN DECLARE SECTION;          /* host variables */
char SQLSTATE[6];
...
EXEC SQL END DECLARE SECTION;
char SQLSTATE_NODATA[6]="02000";
char SQLSTATE_OK[6]="00000";
...
EXEC SQL OPEN get_by_partnum;
...
EXEC SQL FETCH get_by_partnum ... ;      /* Get first row */
while (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    if ... /* Test the value(s) in the current row */
        EXEC SQL DELETE FROM sales.parts /* Delete current row */
            WHERE CURRENT OF get_by_partnum ;
        EXEC SQL FETCH get_by_partnum ... ; /* Get next row */
}
if (strcmp(SQLSTATE, SQLSTATE_NODATA) == 0)
    EXEC SQL CLOSE get_by_partnum;
else
    ... /* Handle other SQLSTATE errors */

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.
01 SQLSTATE-NODATA PIC X(5) VALUE "02000".

```

```

01 SQLSTATE-OK          PIC X(5) VALUE "00000".
...
EXEC SQL OPEN get_by_partnum END-EXEC.
...
EXEC SQL FETCH get_by_partnum ... END-EXEC.
PERFORM UNTIL SQLSTATE = SQLSTATE-NODATA
* Test the value(s) in the current row
  IF ...
    EXEC SQL DELETE FROM sales.parts
      WHERE CURRENT OF get_by_partnum
    END-EXEC.
  END-IF
* Get the next row
  EXEC SQL FETCH get_by_partnum ... END-EXEC.
END-PERFORM.
...
EXEC SQL CLOSE get_by_partnum END-EXEC.

```

Close the Cursor

Use the CLOSE statement to close the cursor and release the result table established by the OPEN statement. After the CLOSE statement executes, the result table no longer exists. To use the same cursor again, you must reopen it by using an OPEN statement.

Note. Ensure that AUTOCOMMIT is off. If AUTOCOMMIT is on, the transaction started by the executor implicitly is committed when the statement is finished.

Use this general syntax:

<pre>CLOSE cursor-name</pre>

For complete syntax, see the CLOSE statement in the *SQL/MX Reference Manual*.

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

Example

This C example closes the `list_by_partnum` cursor:

```
EXEC SQL CLOSE list_by_partnum;
```

The COMMIT WORK and ROLLBACK WORK statements also close all open associated cursors.

To ensure that a sequence of statements that changes the database either executes successfully or not at all, define one transaction consisting of these statements by enclosing the sequence within the BEGIN WORK and COMMIT WORK statements. The CLOSE statement does not save changes. COMMIT WORK saves all changes made in the table. For further information, see [Section 14, Transaction Management](#).

Using Date-Time and INTERVAL Data Types

If a column in the select list of a cursor specification has an INTERVAL or standard date-time (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) data type, use the INTERVAL or date-time type.

If a column in the select list of a cursor specification has a nonstandard SQL/MP DATETIME data type that is not equivalent to DATE, TIME, or TIMESTAMP, you must use the CAST function to convert the column to a character string. You must also specify the length as the length of the target host variable (or the length–1 in the case of a C program) as part of the CAST conversion. Furthermore, if the column in the WHERE clause to be compared to the input value has a nonstandard SQL/MP DATETIME data type, you must use the CAST function to convert the character input value to the appropriate data type.

Standard Date-Time Example

This example shows a typical context for a date-time input parameter for a cursor specification:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4) hv_projcode;
    char                hv_projdesc[19];
    DATE                hv_start_date;
    DATE                in_start_date;
EXEC SQL END DECLARE SECTION;...
EXEC SQL DECLARE get_project CURSOR FOR
    SELECT projcode, projdesc, start_date
    FROM samdbcat.persnl.project
    WHERE start_date <= :in_start_date;

/* Initialize the value in the WHERE clause. */
printf("Enter latest start date in form yyyy-mm-dd: ");
scanf("%s", in_start_date);
/* Open the cursor using this value. */
EXEC SQL OPEN get_project;
/* Fetch the first row of the result table. */
EXEC SQL FETCH get_project
    INTO :hv_projcode, :hv_projdesc, :hv_start_date;

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    /* Process the row in some way. */
    ...
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_project
        INTO :hv_projcode, :hv_projdesc, :hv_start_date;
}
/* Close the cursor. */
EXEC SQL CLOSE get_project;
```

Nonstandard SQL/MP DATETIME Example

This example shows a typical context for a nonstandard date-time input parameter, DATETIME MONTH TO DAY (mm-dd), for a cursor specification:

```

C EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4) hv_projcode;
    char hv_projdesc[19];
    char hv_start_date[6];
    char in_start_date[6];
EXEC SQL END DECLARE SECTION;...
EXEC SQL DECLARE get_project CURSOR FOR
    SELECT projcode, projdesc, CAST(start_date AS CHAR(5))
    FROM samdbcat.persnl.project
    WHERE start_date <=
        CAST(:in_start_date AS DATETIME MONTH TO DAY);

/* Initialize the value in the WHERE clause. */
printf("Enter latest start date in form mm-dd: ");
scanf("%s", in_start_date);
/* Open the cursor using this value. */
EXEC SQL OPEN get_project;
/* Fetch the first row of the result table. */
EXEC SQL FETCH get_project
    INTO :hv_projcode, :hv_projdesc, :hv_start_date;

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    /* Process the row in some way. */
    ...
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_project
        INTO :hv_projcode, :hv_projdesc, :hv_start_date;
}
/* Close the cursor. */
EXEC SQL CLOSE get_project;

```

Interval Example

This example uses a typical context for an interval input parameter for a cursor specification:

```

C EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4) hv_projcode;
    char hv_projdesc[19];
    INTERVAL DAY(4) hv_est_complete;
    INTERVAL DAY(4) in_est_complete;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL DECLARE get_project CURSOR FOR
    SELECT projcode, projdesc, est_complete
    FROM samdbcat.persnl.project
    WHERE est_complete <= :in_est_complete

```

```

/* Initialize the value in the WHERE clause. */
printf("Enter minimum estimated number of days: ");
scanf("%s", in_est_complete);

/* Open the cursor using this value. */
EXEC SQL OPEN get_project;

/* Fetch the first row of the result table. */
EXEC SQL FETCH get_project
    INTO :hv_projcode, :hv_projdesc, :hv_est_complete;

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    /* Process the row in some way. */
    ...
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_project
        INTO :hv_projcode, :hv_projdesc, :hv_est_complete;
}
/* Close the cursor. */
EXEC SQL CLOSE get_project;

```

Using Floating-Point Data Types

You can retrieve floating-point data from SQL/MP and SQL/MX tables to an application program. The type of float returned to the application (IEEE or Tandem) depends on the float host variable that was declared. Based on the declared type, NonStop SQL/MX converts and returns floating-point values to the application in the appropriate form.

If the application directs the SQL/MX preprocessor to declare host variables in IEEE floating-point format and then compiles and links the program with Tandem floating-point format, an application error occurs and incorrect floating point values are returned. This problem also occurs if the host variable is declared as Tandem floating-point and linked as an IEEE floating-point value.

Considerations When Using a Cursor

- If the cursor locks or updates an audited table, the OPEN and FETCH operations and any subsequent cursor operations must be within a transaction.
- A process that uses a cursor must have read authority for tables and views referred to in the SELECT that specifies the cursor. If the cursor declaration specifies FOR UPDATE, the process must also have write authority for the referenced table, view, and base tables underlying the view.

If you use a cursor to locate rows to delete or update without specifying FOR UPDATE in the declaration, NonStop SQL/MX checks only the read authority when the OPEN executes, even though the delete or update requires write authority. NonStop SQL/MX checks for write authority when DELETE or UPDATE executes.

- You can specify **IN EXCLUSIVE MODE** for the **SELECT** statement in the cursor specification so that NonStop SQL/MX does not have to escalate the lock when an **UPDATE** or **DELETE** executes. Otherwise, if you do not specify **IN EXCLUSIVE MODE** and your program is reading records accessed by another cursor defined with **IN EXCLUSIVE MODE**, your program must wait for access.

Cursor Position

Cursor position is similar to record position in a sequential file. An open cursor is positioned either before a certain row, on a certain row, or after the last row.

These operations cause the cursor to be positioned:

OPEN	Before the first row
FETCH	On the retrieved row (the current position)
DELETE	Before the row following the deleted row
UPDATE	No change (the current position)
CLOSE	No position

The **SELECT** statement that specifies the cursor can determine the order in which rows are returned. To specify the order, include an **ORDER BY** clause. Otherwise, the order is undefined.

Cursor Stability

Cursor stability guarantees that a row at the current cursor position cannot be modified by another program, but concurrent access to other rows in the database is allowed. For NonStop SQL/MX to guarantee cursor stability, you can specify **STABLE ACCESS** for the **SELECT** statement that defines the cursor or you can specify **SERIALIZABLE** access.

SERIALIZABLE ACCESS locks the row until the end of the transaction.

STABLE ACCESS is used only for those **SELECT** statements that could potentially be updated. If the shape of a **SELECT** statement is such that it cannot be updated with an updatable cursor (for example it has a join), the **STABLE ACCESS** option is internally changed to **READ COMMITTED** access.

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 query expression in the cursor declaration 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.

For more information, see the *SQL/MX Reference Manual*.

Cursor Sensitivity

The ANSI standard defines three types of cursor sensitivity: INSENSITIVE, SENSITIVE, and ASENSITIVE. NonStop SQL/MX and NonStop SQL/MP provide only ASENSITIVE behavior, which means that the cursor might or might not see the effects of other DML operations that are performed within the same transaction. For example, if you have concurrent DML operations, such as an INSERT, a standalone UPDATE, or standalone DELETE, the result of those operations might not be visible to the cursor. However, any changes you perform by using UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF are visible to the cursor.

7 Static Rowsets

The traditional cursor model in SQL is inefficient for applications retrieving large numbers of rows because too much time is used retrieving one row at a time. However, the SQL/MX extension rowsets enable the SQL cursor to return more than one row at a time, greatly reducing the number of calls made to both the database system and the network.

Rowsets improve the performance of applications requiring simultaneous access to several rows at a time, whether that access is to perform comparisons or other types of processing. Rowsets simplify the task of storing and manipulating a large number of rows in the application address space.

This section describes:

- [What Are Rowsets?](#) on page 7-1
- [Using Rowsets](#) on page 7-2
- [Declaring Host Variable Arrays as Rowsets](#) on page 7-2
- [Specifying Rowset Arrays](#) on page 7-4
- [Using Rowset Arrays in DML Statements](#) on page 7-7
- [Specifying Size and Row ID for Rowset Arrays](#) on page 7-24
- [Specifying Rowset-Derived Tables](#) on page 7-32
- [Using Rowset-Derived Tables in DML Statements](#) on page 7-33

Note. Rowsets are not supported from MXCI.

What Are Rowsets?

The rows returned in a single fetch are called the rowset, and the columns of the rows are the arrays composing the rowset. An application can present a set of column values (rows) in an SQL statement (for example, in the WHERE clauses). The host variable arrays composing a rowset therefore can be used as output variables (for example, in the INTO clause) to receive large amounts of data from SELECT and FETCH statements. Each host variable array receives data from one selected column. Similarly, you can use host variable arrays as input variables for these statements:

- SELECT (WHERE and HAVING clauses)
- INSERT (VALUES clause)
- UPDATE (SET and WHERE clauses)
- DELETE (WHERE clause)

Rowsets are supported for both static and dynamic embedded SQL programs in NonStop SQL/MX. Before using static rowsets, you must declare them as host variable arrays in the SQL DECLARE section by using the keyword ROWSET. For information on dynamic rowsets, see [Section 12, Dynamic SQL Rowsets](#).

Note. Examples in this section show how to use rowsets from an SQL/MX program that accesses SQL/MX and SQL/MP tables. Rowsets as a feature do not exist in NonStop SQL/MP.

Using Rowsets

The two ways of using rowsets in SQL queries are:

- Direct use. You can place host variable rowset arrays anywhere a scalar host variable is placed in an SQL query.
- Rowset-derived tables. Given a rowset, a construct is provided that creates an in-memory table. A rowset-derived table resulting from this construct is a table of several columns (one column for each array of the rowset) and rowset size tuples or rows.

A rowset is analogous to an in-memory table. A rowset with one host variable array of n elements is similar to a temporary in-memory table with n tuples, where the j element value of the array corresponds to the j tuple of the table. A rowset with M host variable arrays of n elements is similar to a temporary in-memory table with M columns and n tuples. The j element value of array A corresponds to the A column of the j tuple of the table. Within the scope of an SQL statement and by using rowset-derived table syntax, you can create and use in-memory tables in a way similar to ordinary tables.

An SQL statement containing a rowset of size n for input is handled like a join of the tables composing the SQL statement with the rowset or the execution of the same statement n times using successive elements of the rowset. The semantics and side effects of rowsets are explained assuming that the rowset is just another table in the SQL statement.

For output, a cursor is typically needed in a SELECT statement unless the SELECT statement returns a single row. When you use rowset arrays as host variables to retrieve results, this rule is expanded as follows:

- A cursor is not needed if the SELECT statement returns no more than the size of the rowset.
- A cursor must be used when the maximum number of rows cannot be estimated or when memory requirements are too high to store the result table. In this case, the result table is retrieved in rowset size batches using the FETCH statement.

Declaring Host Variable Arrays as Rowsets

A host variable array, along with its dimension, is declared within the SQL Declare Section of an embedded SQL program. A rowset array is a host variable array that is declared for each column in a query. A rowset consists of a collection of rowset arrays. Each rowset array contains as many elements as there are in the rowset.

The dimensions of the arrays that make up a rowset correspond to the desired number of elements. All arrays must have the same number of dimensions as the other arrays in the rowset or be at least as large as the desired rowset. NonStop SQL/MX uses the smallest dimension as the rowset size while performing operations into the rowset.

To specify a host variable array as a part of a rowset, use this syntax:

```
ROWSET [rowset-size] variable-specification
```

rowset-size

specifies the dimension of the host variable array that is a part of the rowset. The size immediately follows the ROWSET keyword and must be enclosed in square brackets []. The size is an unsigned integer.

variable-specification

is the data type and name of a host variable. It can be any valid host language identifier with a data type that corresponds to an SQL data type. For information on SQL data types and C host variable data types, see [Table 3-4](#) on page 3-12. For the COBOL equivalent, see [Table 4-2](#) on page 4-7.

Note. In the examples in this section, note that COBOL references the rowset array as elements 1 through 5. (C references the array as elements 0 through 4.) However, when NonStop SQL/MX references the COBOL array as a derived table, those elements are referenced as rows 0 through 4.

Example

This example uses three arrays of 200 elements, which are used to retrieve at most 200 rows of a table. You can use the fourth array as an indicator array for the salary:

C

```
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET [200] char          hva_first_name[16];
ROWSET [200] char          hva_last_name[21];
ROWSET [200] unsigned NUMERIC (8,2) hva_salary;
ROWSET [200] short         hva_salary_indicator;
...
EXEC SQL END DECLARE SECTION;
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
02 ROWSET[200] hvafirstname pic x(15).
02 ROWSET[200] hvalastname  pic x(20).
02 ROWSET[200] hvasalary    pic 9(8)v9(2) comp.
02 ROWSET[200] hvasalaryindicator pic s9(4) comp.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

Rowset Host Variable Pointers

A rowset host variable cannot be declared as a pointer. If you need to access a rowset host variable through a pointer, you must make the rowset host variable a structure field.

Example

The following example shows an improper usage of rowset host variable where the rowset host variable is declared as a pointer type:

```
EXEC SQL BEGIN DECLARE SECTION;
/* cannot declare a rowset pointer as host variable */
ROWSET [100] int * rowsetPtr;
...
```

The following example shows the correct use of rowsets in a structure:

```
EXEC SQL BEGIN DECLARE SECTION;
struct ptrType
{
    /* rowsets are allowed as structure fields */
    ROWSET [100] int rowsetField;
    ...
};
struct ptrType * structPtr;
...
```

Considerations for Rowset Size

- The total rowset size (that is, the size of the row times the number of rows) should not produce fragmentation in the network or process communication.
- The total rowset size should not exceed the physical memory of the client computer to avoid fragmentation while accessing a rowset array.
- The rowset size should not be less than the number of rows that need to be accessed simultaneously. For example, a screen-based application should use a rowset size that is a multiple of the number of rows displayed on the screen.

Specifying Rowset Arrays

After you declare a host variable array that is a part of a rowset, use this syntax to specify it within an embedded SQL statement.

```
:array-name [[INDICATOR] :indicator-array-name]
```

array-name

is the host variable array name. It can be any valid host language identifier with a data type that corresponds to an SQL data type. You must precede *array-name* with a colon (:) within an SQL statement.

INDICATOR

is an optional keyword that precedes *indicator-array-name*.

indicator-array-name

is an indicator variable array of exact numeric data type. This data type is `short` in C or `PIC 9(4) comp` in COBOL. You must precede *indicator-array-name* with a colon (:) in an SQL statement.

If data returned in the host variable array for a particular row and column is null, the corresponding indicator variable is set to `-1`. If character data returned is truncated, the indicator variable is set to the length of the string in the database. Otherwise, the value of the indicator variable is zero. To insert null into the database, set the indicator variable to a value less than zero for a particular row and column in the corresponding host variable array. For inserting nonnull values, the corresponding indicator variable must be set to zero. This last rule is also true for all input arrays (for example, those used in WHERE and SET clauses). You generate a run-time error if you specify a positive value in an indicator for input.

Example

This example retrieves three columns of, at most, 200 rows of a table. The salary column can be null, and the salary array is followed by an indicator array:

C

```
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET [200] char          hva_first_name[16];
ROWSET [200] char          hva_last_name[21];
ROWSET [200] unsigned NUMERIC (8,2) hva_salary;
ROWSET [200] short         hva_salary_indicator;
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL
SELECT first_name, last_name, salary
INTO :hva_first_name, :hva_last_name,
:hva_salary INDICATOR :hva_salary_indicator
FROM persnl.employee;
```

COBOL

```
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
02 ROWSET[200] hvafirstname pic x(15).
02 ROWSET[200] hvalastname  pic x(20).
02 ROWSET[200] hvasalary    pic 9(8)v9(2) comp.
02 ROWSET[200] hvasalaryindicator pic 9(5).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL
SELECT first_name, last_name, salary
INTO :hvafirstname, :hvalastname,
:hvasalary INDICATOR :hvasalaryindicator
```

```
FROM employee END-EXEC.  
...
```

Using Rowset Arrays for Input

Use array host variables as input in the WHERE clause of SELECT, DELETE, or UPDATE statements, the HAVING clause of a SELECT statement, the SET clause of an UPDATE statement, and the VALUES clause of an INSERT statement.

Use rowset arrays for input to provide a looping mechanism equivalent to multiple logical executions of the same SQL statement, once for each set of input values. However, diagnostic information is returned only for the whole SQL statement and not for each set of input values. When you use more than one array host variable as input in an SQL statement, the input arrays might not all be of uniform size. In this situation, the number of input values is equal to the size of the smallest input array. If you use scalar host variables along with some rowset arrays as input, the scalar values are duplicated as many times as the size of the smallest input array. This scenario is semantically equivalent to replacing the scalar input host variable with an array (whose size is the same as the smallest input array in that SQL statement), every element of which has the same value as the scalar host variable.

When you use rowset arrays as input, check that all array elements up to the size of the smallest input array have valid values. If not all elements of the input array are used, specify the size of the input array size to be a smaller value by using the ROWSET FOR INPUT SIZE syntax. See [Specifying Size and Row ID for Rowset Arrays](#) on page 7-24.

Using Rowset Arrays for Output

Use array host variables for output in the INTO clause of SELECT and FETCH statements.

Use rowset arrays for output to retrieve multiple rows from the result table by executing a single SQL statement. When more than one array host variable is used as output in an SQL statement, the output arrays might not be of uniform size. In this situation, the number of output rows retrieved is equal to the size of the smallest output array.

When you use the SELECT INTO statement, check that the number of rows in the result table is not larger than the size of the smallest output array. If the result table is larger than the output array size, you must declare a cursor, and the FETCH statement must be executed multiple times to retrieve all the rows in the result table.

Do not use rowset arrays for output in a cursor declaration or with dynamic rowsets.

Using Rowset Arrays in DML Statements

Technique	Description
Selecting Rows Into Rowset Arrays	Multiple rows of data are retrieved from a table or a view, and the specified column values are placed into host variable arrays. Multiple search conditions can also be specified by using host variable arrays in the WHERE clause.
Inserting Rows From Rowset Arrays	Multiple rows are inserted into a table or view by using arrays of values in the VALUES clause of an INSERT statement.
Updating Rows by Using Rowset Arrays	Multiple logical executions of an UPDATE statement are performed by using arrays of values in the SET and WHERE clause.
Deleting Rows by Using Rowset Arrays	Multiple logical executions of the DELETE statement are performed by using arrays of values in the WHERE clause.

Note. This list of where you can use rowset arrays is not exhaustive. In general, wherever you specify a scalar host variable in an SQL statement, you can substitute a rowset array host variable of equivalent type.

Selecting Rows Into Rowset Arrays

Use a SELECT INTO statement using a rowset as output to retrieve multiple rows of data from one or more tables or views and place column values into corresponding host variable arrays. The set of rows returned in a single SELECT statement is called the rowset, and the columns of the rows are the arrays composing the rowset.

Use this general syntax:

```
SELECT column [,column]...
INTO :hostvar-array [,:hostvar-array]...
FROM table-name [,table-name]...
[WHERE search-condition]
[GROUP BY column [,column]...]
[HAVING search-condition]
[ORDER BY column [,column]...]
```

For complete syntax, see the SELECT statement in the *SQL/MX Reference Manual*.

Note. Data mining operations—SAMPLE, SEQUENCE BY, and TRANSPOSE—are not supported for operations with rowsets. Some Publish/Subscribe operations are not supported with rowsets either. Specifically, you cannot use rowsets as input (in WHERE and SET clauses) with embedded UPDATES and DELETES. Additionally, you cannot join a rowset-derived table with an embedded UPDATE or DELETE.

Example

This example uses a SELECT statement returning an employee's first name, last name, and department from the EMPLOYEE table. The elements in the target host variable arrays are in the order based on the columns in the ORDER BY clause.

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET [100] unsigned NUMERIC (4)   hva_deptnum;
    ROWSET [100] char                   hva_firstname[16];
    ROWSET [100] char                   hva_lastname[21];
    ...
    long  numrows;
EXEC SQL END DECLARE SECTION;
long i;
...
EXEC SQL
    SELECT first_name, last_name, deptnum
    INTO :hva_firstname, :hva_lastname, :hva_deptnum
    FROM persnl.employee
    ORDER BY deptnum, last_name, first_name;
...
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
...
for (i = 0; i < numrows; i++) {
    /* NOTE: The null termination can also be done */
    /* before the SELECT statement.                */

    hva_firstname[i][15] = '\0';
    hva_lastname[i][20] = '\0';
    printf("\nDept: %hu, Name: %s, %s",
        hva_deptnum[i], hva_lastname[i], hva_firstname[i]);
    ...
    /* Process the row in some way. */
    ....
}
}
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 numrows                  pic 9(9) comp.
01 rs.
    02 ROWSET[100] hvadeptnum    pic 9(4) comp.
    02 ROWSET[100] hvafirstname  pic x(15).
    02 ROWSET[100] hvalastname   pic x(20).
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                        pic s9(4) comp.
...
EXEC SQL
    SELECT first_name, last_name, deptnum
    INTO :hvafirstname, :hvalastname, :hvadeptnum
    FROM employee
    ORDER BY deptnum, last_name, first_name END-EXEC.
EXEC SQL
```



```

GET DIAGNOSTICS :numrows = ROW_COUNT end-exec.
PERFORM VARYING i FROM 1 BY 1 UNTIL i > numrows
  display "Dept: " hvadeptnum(i) "Name: " hvalastname(i)
    hvafirstname(i)
end-perform.
...

```

The previous example is correct only if the SELECT INTO statement is certain to return fewer than 100 rows. If more than 100 rows are returned, an SQLSTATE value is returned.

If the SELECT statement can return more rows than are allocated in the rowset array, you have these choices:

- Limit the SQL query so that it returns only a specified number of rows, as shown:

```

...
EXEC SQL
  SELECT [first 100]first_name, last_name, deptnum
  INTO :hva_firstname, :hva_lastname, :hva_deptnum
  FROM persnl.employee
  ORDER BY deptnum, last_name, first_name;
...

```

- Use a rowset cursor to get all the results from the SELECT statement. See [Selecting Rowsets With a Cursor](#) on page 7-16.

Getting the Number of Retrieved Rows

In the preceding example, the actual number of rows retrieved is stored in the diagnostics area. NonStop SQL/MX stores completion and exception information for an embedded SQL statement in this area. NonStop SQL/MX automatically allocates the diagnostics area in a program. You are not required to explicitly allocate it yourself.

At the beginning of the execution of an SQL statement, the diagnostics area is emptied. When the statement executes, NonStop SQL/MX places information on completion or exception conditions into this area. The diagnostics area consists of:

- Statement information: Header area with information on the SQL statement as a whole
- Condition information: Detail area with information on each error, warning, or completion code that occurs during the execution of the SQL statement

The number of retrieved rows is stored in the ROW_COUNT field of the statement information in the diagnostics area. You can retrieve the value in the ROW_COUNT field by using the GET DIAGNOSTICS statement. In the preceding example, the statement that retrieves the value of ROW_COUNT is specified as:

```
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
```

For further information, see the GET DIAGNOSTICS statement in the *SQL/MX Reference Manual*.

Retrieving the Row Number for a Failed Operation

NonStop SQL/MX returns an error when an INSERT or UPDATE operation using a rowset fails. The row number causing the error can be retrieved.

NonStop SQL/MX stores completion and exception information for an embedded SQL statement in the diagnostics area. NonStop SQL/MX automatically allocates this area in memory.

The diagnostics area is empty before you execute an SQL statement. When you execute the statement, NonStop SQL/MX places the information about completion or exception conditions into the diagnostics area. The diagnostics area consists of the following:

- **Statement Information:** The header area that contains information about the SQL statement.
- **Condition Information:** The detail area that contains information on errors, warnings, or completion codes that occur during the execution of the SQL statement.

The row number that causes the error in an INSERT or UPDATE operation is stored in the field `ROW_NUMBER` in the statement information of the diagnostics area. You can retrieve the value in the field `ROW_NUMBER` by using the `GET DIAGNOSTICS` statement.

For more information, see the `GET DIAGNOSTICS` statement in the *SQL/MX Release 3.2 Reference Manual*.

Example

The following example retrieves the row number that causes the error in the insert operation which is stored in the `ROW_NUMBER` field of the diagnostics area:

```
DDL: create table rownum1(a int, b int, primary key(a));
```

Program:

```

exec sql whenever sqlerror call display_diagnosis;

EXEC SQL BEGIN DECLARE SECTION;
    ROWSET [10] long g_int;
    ROWSET [10] long h_int;
EXEC SQL END DECLARE SECTION;

int main(){
printf("\n\ntest1 : Expecting rownumber = 9\n");
int i=0;
for (i=0; i<10; i++) {
    g_int[i] = i;
    h_int[i] = i;
}

    g_int[9] = 7; /* causes unique constraint error */
    h_int[5] = 3;

EXEC SQL DELETE FROM rownum1;
EXEC SQL  INSERT INTO rownum1 VALUES (:g_int, :h_int) ;
if (SQLCODE != 0) {
    printf("Failed to insert. SQLCODE = %ld\n",SQLCODE);
}
else {
    printf("Insert succeeded. SQLCODE = %ld\n", SQLCODE);
    EXEC SQL COMMIT ;
}
}

```

```

void display_diagnosis()
{
    exec sql get diagnostics :hv_num = NUMBER;
    memset(hv_msgtxt, ' ', sizeof(hv_msgtxt));
    hv_msgtxt[512]='\0';
    printf("Number   : %d\n", hv_num);
    for (i = 1; i <= hv_num; i++) {
        exec sql get diagnostics exception :i
            :hv_sqlcode = SQLCODE,
/* this gets the row number which is causing the unique
constraint error */
        :hv_rownum    = ROW_NUMBER,
        :hv_msgtxt    = MESSAGE_TEXT;
        printf("Sqlcode  : %d\n", hv_sqlcode);
        printf("Message  : %s\n", hv_msgtxt);
        printf("RowNum   : %d\n", hv_rownum);
    }
}

```

The output for the example is as follows:

```
test1: Expecting rownumber = 9
```

```
Number   : 1
```

```
Sqlcode  : -8102
```

```
Message  : *** ERROR[8102] The operation is prevented by a unique
constraint.
```

```
RowNum   : 9
```

```
Failed to insert.  SQLCODE = -8102
```

Selecting a Column With Date-Time or INTERVAL Data Type

If a column in the select list has an INTERVAL or standard date-time (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) data type, use the INTERVAL or date-time data types.

If a column in the select list has a nonstandard SQL/MP DATETIME data type that is not equivalent to DATE, TIME, or TIMESTAMP, use the CAST function to convert the column to a character string. You must also specify the length of the target host variable (or the length–1 in the case of a C program) in the AS clause of the CAST conversion.

For more information on declaring date-time or INTERVAL data types, see [Section 3, Host Variables in C/C++ Programs](#) and [Section 4, Host Variables in COBOL Programs](#).

Examples

This example uses a typical context for selecting a standard date-time value. The number of rows in the PROJECT table with a start date less than or equal to 1998-12-01 does not exceed 200:

C

```
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET [200] unsigned NUMERIC (4)   hva_projcode;
ROWSET [200] char                    hva_projdesc[19];
ROWSET [200] DATE                    hva_start_date;
...
long numrows;
EXEC SQL END DECLARE SECTION;
long i;
...
EXEC SQL
SELECT projcode, projdesc, start_date
INTO :hva_projcode, :hva_projdesc, :hva_start_date
FROM persnl.project
WHERE start_date <= DATE '1998-12-01';
...
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
...
for (i = 0; i < numrows; i++) {
    hva_projdesc[i][18] = '\0';
    hva_start_date[i][10] = '\0';
    printf("\nProject: %hu, %s, Started: %s",
        hva_projcode[i], hva_projdesc[i], hva_start_date[i]);
    ...
    /* Process the row in some way. */
    ....
}
```

COBOL

```
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 numrows                 pic 9(9) comp.
01 rs.
    02 ROWSET[200] hvaprojcode    pic 9(4) comp.
    02 ROWSET[200] hvaprojdesc    pic x(18).
    02 ROWSET[200] hvastartdate    DATE.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                      pic s9(4) comp.
...
EXEC SQL
```

```
SELECT projcode, projdesc, start_date
INTO :hvaprojcode, :hvaprojdesc, :hvastartdate
FROM project
WHERE start_date <= DATE '1998-12-01' END-EXEC.
EXEC SQL
  GET DIAGNOSTICS :numrows = ROW_COUNT end-exec.
PERFORM VARYING i FROM 1 BY 1 UNTIL i > numrows
  display "Project:  " hvaprojcode(i) hvaprojdesc(i)
                                "Started: " hvastartdate(i)
END-PERFORM
...
```

Rowset Arrays as Input for SELECT Statements

A SELECT statement retrieves the values in one or more columns of the matching rows. The matching rows are determined by the evaluation of the search condition in the WHERE clause of the SELECT statement.

You can use rowset arrays as input in the WHERE clause search condition to specify multiple values for the search condition in a single SQL statement. The use of rowset arrays for input is similar to a looping mechanism whereby the same statement is executed multiple times with a different set of values for input each time. You can use rowset arrays as input in the HAVING clause search condition of a SELECT statement.

Example

This example selects the EMPNUM and SALARY columns of all rows in the EMPLOYEE table where the (JOBCODE, DEPTNUM) value is equal to one of the set of values in the hva_jobcode and hva_deptnum host variable arrays. An input value set is composed of array elements from the hva_jobcode and hva_deptnum host variable arrays with identical index numbers. Five input value sets exist, and the SELECT statement is executed for each matching input value set:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[5] unsigned NUMERIC (4)      hva_jobcode;
    ROWSET[5] unsigned NUMERIC (4)      hva_deptnum;
    ROWSET [100] unsigned NUMERIC (4)    hva_empnum;
    ROWSET [100] unsigned NUMERIC (8,2)  hva_salary;
    ROWSET [100] short                   hva_salary_indicator;
    ...
    long numrows;
EXEC SQL END DECLARE SECTION;

...
/* Populate the jobcode and deptnum rowsets in some way. */
hva_jobcode[0] = 100;
hva_deptnum[0] = 9000;
hva_jobcode[1] = 200;
hva_deptnum[1] = 9000;
hva_jobcode[2] = 300;
hva_deptnum[2] = 1000;
hva_jobcode[3] = 400;
hva_deptnum[3] = 1000;
hva_jobcode[4] = 500;
hva_deptnum[4] = 3000;
...
EXEC SQL
    SELECT empnum, salary
    INTO :hva_empnum,
        :hva_salary INDICATOR :hva_salary_indicator
    FROM persnl.employee
    WHERE jobcode = :hva_jobcode AND deptnum = :hva_deptnum;
...

```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 numrows                  pic 9(9) comp.
01 rs.
    02 ROWSET[5] hvajobcode    pic 9(4) comp.
    02 ROWSET[5] hvadeptnum    pic 9(4) comp.
    02 ROWSET[100] hvaempnum    pic 9(4) comp.
    02 ROWSET[100] hvasalary    pic 9(8)v9(2) comp.
    02 ROWSET[100] hvasalaryindicator pic s9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.

...
**** populate the jobcode and deptnum rowsets in some way ****
Move 100 TO hvajobcode(1)
Move 9000 TO hvadeptnum(1)
Move 200 TO hvajobcode(2)

```

```

Move 9000 TO hvadeptnum(2)
Move 300 TO hvajobcode(3)
Move 1000 TO hvadeptnum(3)
Move 400 TO hvajobcode(4)
Move 1000 TO hvadeptnum(4)
Move 500 TO hvajobcode(5)
Move 3000 TO hvadeptnum(5)
EXEC SQL
    SELECT empnum, salary
    INTO :hvaempnum, :hvasalary INDICATOR :hvasalaryindicator
    FROM employee
    WHERE jobcode = :hvajobcode AND
          deptnum = :hvadeptnum
END-EXEC.
...

```

Selecting Rowsets With a Cursor

If the number of rows returned by a SELECT statement exceeds the size of the rowset array, use a FETCH statement with a rowset cursor to cycle over a specific number of rows. The next example uses a rowset cursor to fetch and print the project code, the project description, and the start date of all projects started before a specific date.

For complete syntax, see the FETCH statement in the *SQL/MX Reference Manual*.

Examples

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET[200] unsigned NUMERIC (4)      hva_projcode;
ROWSET[200] char                       hva_projdesc[19];
ROWSET[200] DATE                       hva_start_date;
long                                    SQLCODE;
long                                    numrows;
EXEC SQL END DECLARE SECTION;
long i;

/* null terminate char arrays */
for (i = 0; i < 200; i++) {
    hva_projdesc[i][18] = '\0';
    hva_start_date[i][10] = '\0';
}
/* declare cursor for select operation */
EXEC SQL
    DECLARE rowset_cursor CURSOR FOR
    SELECT projcode, projdesc, start_date
    FROM persnl.project
    WHERE start_date <= DATE '1998-12-01';

/* open the cursor */
EXEC SQL
    OPEN rowset_cursor;

/* Fetch all rows of the result table */
WHILE (SQLCODE == 0) {
    EXEC SQL
        FETCH rowset_cursor
        INTO :hva_projcode, :hva_projdesc, :hva_start_date;
    IF ((SQLCODE == 0) || (SQLCODE == 100)) {
        EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
    }
}

```



```

        IF (SQLCODE != 0) {
            printf("GET DIAGNOSTICS operation failed."
                " SQLCODE = %ld\n", SQLCODE);
            return(SQLCODE);
        }
        for (i = 0; i < numRows; i++) {
            printf("Project Code = %s\t
                Project Description = %s\t
                Start Date = %s\n", hva_projcode[i],
                hva_projdesc[i], hva_start_date[i]);
        }
    }
}
/* Close the cursor */
EXEC SQL
    CLOSE rowset_cursor ;
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLCODE                                pic s9(9) comp.
01 numRows                                pic 9(9) comp.
01 rs.
    02 ROWSET[200] hvaprojcode            pic 9(4) comp.
    02 ROWSET[200] hvaprojdesc            pic x(18).
    02 ROWSET[200] hvastartdate            DATE.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                                      pic s9(4) comp.
...
**** declare cursor for select operation ****

EXEC SQL
    DECLARE rowset_cursor CURSOR FOR
    SELECT projcode, projdesc, start_date
    FROM persnl.project
    WHERE start_date <= DATE '1998-12-01'
END-EXEC.
**** open the cursor ****
EXEC SQL
    OPEN rowset_cursor
END-EXEC.
**** Fetch all rows from result table ****
Perform until sqlcode not equal 0
    EXEC SQL FETCH rowset_cursor
        INTO :hwaprojcode, :hwaprojdesc, :hvastartdate
    END-EXEC.
    if SQLCODE EQUAL 0
        EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT END-EXEC.
        if SQLCODE NOT EQUAL 0
            Display "GET DIAGNOSTICS operation failed."
        end-if
        Perform varying i from 1 by 1 until i > numRows
            Display "ProjectCode = " hvaprojcode(i)
            Display "Project Description = " hvaprojdesc(i)
            Display "Start Date = " :hvastartdate(i)
        end-perform.
    end-if
end-perform.
**** Close the cursor ****
EXEC SQL
    CLOSE rowset_cursor
END-EXEC.
...

```

Inserting Rows From Rowset Arrays

The INSERT statement using rowsets inserts multiple rows into a table from host variable arrays. To insert data, a program moves the new values to the array of host variables that have been declared as rowsets and then executes an INSERT statement to transfer the values from the host variable arrays to the table.

Use this general syntax:

```
INSERT INTO table-name [(column [,column]...)]
VALUES (:hostvar-array [, :hostvar-array]...)
```

For complete syntax, see the INSERT statement in the *SQL/MX Reference Manual*.

Example

This example inserts multiple rows (JOBCODE and JOBDESC columns) from host variable arrays into the JOB table:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[5] unsigned NUMERIC (4)   hva_jobcode;
    ROWSET[5] VARCHAR                 hva_jobdesc[19];
    ...
    long  numrows;
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
strcpy(hva_jobdesc[0], "PROJECT MANAGER");
hva_jobcode[1] = 200;
strcpy(hva_jobdesc[1], "PROGRAM MANAGER");
hva_jobcode[2] = 300;
strcpy(hva_jobdesc[2], "QUALITY SUPERVISOR");
hva_jobcode[3] = 400;
strcpy(hva_jobdesc[3], "TECHNICAL OFFICER");
hva_jobcode[4] = 500;
strcpy(hva_jobdesc[4], "EXECUTIVE OFFICER");
...
EXEC SQL INSERT INTO persnl.job (jobcode, jobdesc)
        VALUES (:hva_jobcode, :hva_jobdesc);
...
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 numrows                  pic 9(9) comp.
01 rs.
    02 ROWSET[5]  hvajobcode  pic 9(4) comp.
    02 ROWSET[5]  hvajobdesc  pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.
...
***** Populate the rowset in some way *****
Move 100 to hvajobcode[1]
Move "PROJECT MANAGER" to hvaprojdesc[1]
Move 200 to hvajobcode[2]
```

```

Move "PROGRAM MANAGER" to hvaprojdesc[2]
Move 300 to hvajobcode[3]
Move "QUALITY SUPERVISOR" to hvaprojdesc[3]
Move 400 to hvajobcode[4]
Move "TECHNICAL OFFICER" to hvaprojdesc[4]
Move 500 to hvajobcode[5]
Move "EXECUTIVE OFFICER" to hvaprojdesc[5]
EXEC SQL INSERT INTO job (jobcode, jobdesc)
      VALUES (:hvajobcode, :hvajobdesc) END-EXEC.
...

```

Using Arrays in Expressions and Functions

Unless the arrays are in an INTO clause or in a rowset-derived table, on rowsets use:

- Numeric value expressions and functions
- Character value expressions and functions

The guideline is, wherever you can use a host variable in an expression or function, you can use a rowset array if it is for input (for example, WHERE clause, VALUES clause, SET clause).

When array expressions involve binary arithmetic operators, the two possibilities for the operand types are:

- Array and a constant. For example, an array multiplied by a constant. The semantics are that every element in the array is multiplied by the constant.
- Array and an array. For example, an array multiplied by another array. In this case, both arrays have to be the same length. The semantics are that the first element of array1 is multiplied with the first element of array2, second element of array1 by second element of array2, and so on.

Inserting Null

You can insert multiple rows of data with a null value for one of the columns in some of the rows by using an indicator host variable array.

Example

This example inserts 100 rows into the EMPLOYEE table and sets the SALARY column to null for the first 50 rows by using an indicator host variable array. They use –1 value as the null indicator. For the remaining 50 rows, they set the SALARY column to nonnull values. The indicator host variable array must contain the value 0 for these rows.

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET [100] unsigned NUMERIC (4)   hva_empnum;
ROWSET [100] char                   hva_first_name[16];
ROWSET [100] char                   hva_last_name[21];
ROWSET [100] unsigned NUMERIC (4)   hva_deptnum;
ROWSET [100] unsigned NUMERIC (4)   hva_jobcode;

```

```

ROWSET [100] unsigned NUMERIC (4)   hva_salary;
ROWSET [100] short                   hva_salary_indicator;
...
EXEC SQL END DECLARE SECTION;
long i;
...
/* Populate the host variable arrays in some way. */
...

/* Store -1 in the indicator array for the first 50 input
values. */
for (i = 0; i < 50; i++) hva_salary_indicator[i] = -1;
/* Store 0 in the indicator array for the next 50 input values.
It is assumed that there are valid values for salary in the
hva_salary rowset array from element no. 50 up to element no. 99
*/
for (i = 50; i < 100; i++) hva_salary_indicator[i] = 0;
EXEC SQL
    INSERT INTO persnl.employee
    VALUES ( :hva_empnum,:hva_first_name,
              :hva_last_name,:hva_deptnum,:hva_jobcode,
              :hva_salary INDICATOR :hva_salary_indicator);
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[100] hvaempnum      pic 9(4) comp.
    02 ROWSET[100] hvafirstname   pic x(15).
    02 ROWSET[100] hvalastname    pic x(20).
    02 ROWSET[100] hvadeptnum     pic 9(4) comp.
    02 ROWSET[100] hvajobcode     pic 9(4) comp.
    02 ROWSET[100] hvasalary      pic 9(4) comp.
    02 ROWSET[100] hvasalaryindicator pic s9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                        pic s9(4) comp.
...
**** Populate the host variables arrays in some way ****
**** Store -1 in the indicator array for the first 50 ****
**** input values. ****
PERFORM VARYING i FROM 1 BY 1 UNTIL i = 50
    Move -1 to hvasalaryindicator(i)
end-perform.

**** Store 0 in the indicator array for the next 50 ****
**** input values. It is assumed that there are valid ****
**** values for salary in the hvasalary rowset array ****
**** from element no. 51 upto element no. 100 ****
PERFORM VARYING i FROM 51 BY 1 UNTIL i = 100
    Move 0 to hvasalaryindicator(i)
end-perform.

EXEC SQL

```

```

INSERT INTO employee
VALUES ( :hvaempnum, :hvafirstname, :hvalastname,
        :hvadeptnum, :hvajobcode,
        :hvasalary INDICATOR :hvasalaryindicator)
END-EXEC.

```

...

Inserting a Timestamp Value

You do not need to use the CAST function when inserting a TIMESTAMP value.

Example

This example inserts multiple rows into the PROJECT table, including a TIMESTAMP value in the SHIP_TIMESTAMP column:

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET [10] TIMESTAMP      hva_timestamp;
...
EXEC SQL END DECLARE SECTION;
long i;
...
/* Populate the host variable arrays in some way. */
...
EXEC SQL INSERT INTO PROJECT
(..., SHIP_TIMESTAMP, ...)
VALUES(..., :hva_timestamp, ...);
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                      pic x(5).
01 rs.
    02 ROWSET[10]  hvaempnum      pic 9(4) comp.
    02 ROWSET[10]  hvaprojcode    pic 9(4) comp.
    02 ROWSET[10]  hvatimestamp   TIMESTAMP.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                            pic s9(4) comp.
...
**** Populate the host variables arrays in some way ****
EXEC SQL INSERT INTO project
( empnum,projcode,ship_timestamp)
VALUES (:hvaempnum,:hvaeprojcode, :hvatimestamp) END-EXEC.
...

```

Updating Rows by Using Rowset Arrays

The searched UPDATE statement updates the values in one or more columns of the matching rows of a table or view. The matching rows are determined by the evaluation of the search condition in the WHERE clause of the UPDATE statement. You can perform multiple logical executions of the statement by using arrays of values in the WHERE clause. Use of array host variables in the SET clause is optional.

Use this general syntax:

```
UPDATE table-name
SET set-clause [,set-clause]. . .
WHERE search-condition
```

set-clause

The expression in a *set-clause* can contain array host variables. When array host variables are present in the *search-condition*, two alternatives exist for the *set-clause* expression:

- Scalar host variables only. In this case, all matching rows are updated with identical values, obtained by evaluating the scalar expression. This case is shown in the next example.
- Some array host variables. You can use a rowset in the SET clause only if you have a rowset in the WHERE clause. If the size of the rowsets are not the same in the SET and WHERE clauses, the smaller of the two sizes are used and all rowset elements beyond the smaller size are ignored. All rows returned due to the first element in the *search-condition* array are updated using the value obtained by evaluating the first element in the *set-clause* array. All matching rows due to the second element in the *search-condition* array are updated using the second element in the *set-clause* array, and so on.

search-condition

must contain host variable arrays if you use rowsets in an UPDATE statement. The use of rowset arrays for input is similar to a looping mechanism whereby the same statement is executed multiple times with a different set of values for input each time.

For complete syntax, see the UPDATE statement in the *SQL/MX Reference Manual*.

Example

This example updates the SALARY column of all rows in the EMPLOYEE table where the JOBCODE value is equal to one of the values in the `hva_jobcode` host variable array. The UPDATE statement is executed for each matching job code:

```
C EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET[5] unsigned NUMERIC (4)  hva_jobcode;
unsigned NUMERIC (4,2) hv_inc;
. . .
long  numrows;
EXEC SQL END DECLARE SECTION;
. . .
/* Input the salary increment. */
. . .
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
```

```

hva_jobcode[2] = 300;
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;
...
EXEC SQL UPDATE persnl.employee
      SET salary = salary * :hv_inc
      WHERE jobcode = :hva_jobcode;
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 sqlstate                pic x(5).
  01 numrows                  pic 9(9) comp.
  01 rs.
  01 hva-multiplier           pic 9(4) comp.
    02 ROWSET[5]  hvajobcode pic 9(4) comp.
    02 ROWSET[5]  hvainc     pic 9(4)v99 comp.
EXEC SQL END DECLARE SECTION END-EXEC.
...

****Input the salary increment ****
****Populate the rowset in some way ****

Move 100 TO hvajobcode (1)
Move 200 TO hvajobcode (2)
Move 300 TO hvajobcode (3)
Move 400 TO hvajobcode (4)
Move 500 TO hvajobcode (5)
EXEC SQL UPDATE employee
      SET salary = salary * :hvainc
      WHERE jobcode = :hvajobcode
END-EXEC.
...

```

The number of updated rows is stored in the ROW_COUNT field of the statement information in the diagnostics area. You can retrieve the value in the ROW_COUNT field by using the GET DIAGNOSTICS statement.

Deleting Rows by Using Rowset Arrays

The searched DELETE statement deletes matching rows of a table or view. The matching rows are determined by the evaluation of the search condition in the WHERE clause of the DELETE statement. Multiple logical executions of the statement are performed by using arrays of values in the WHERE clause.

Use this general syntax:

```

DELETE FROM table-name
WHERE column = :hostvar-array

```

For complete syntax, see the DELETE statement in the *SQL/MX Reference Manual*.

Example

This example deletes all rows from the JOB table specified by the `hva_jobcode` host variable array:

```
C EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[5] unsigned NUMERIC (4)  hva_jobcode;
    ...
    long  numrows;
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 300;
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;
...
EXEC SQL
    DELETE FROM persnl.job
    WHERE jobcode = :hva_jobcode;
...
```

The number of deleted rows is stored in the `ROW_COUNT` field of the statement information in the diagnostics area. You can retrieve the value in the `ROW_COUNT` field by using the `GET DIAGNOSTICS` statement.

```
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 sqlstate                pic x(5).
    01 numrows                  pic 9(9) comp.
    01 rs.
        02 ROWSET[5]  hvajobcode pic 9(4) comp.
    EXEC SQL END DECLARE SECTION END-EXEC.
****Populate the rowset in some way ****
    Move 100 TO hvajobcode (1)
    Move 200 TO hvajobcode (2)
    Move 300 TO hvajobcode (3)
    Move 400 TO hvajobcode (4)
    Move 500 TO hvajobcode (5)

    EXEC SQL DELETE FROM job
        WHERE jobcode = :hvajobcode
END-EXEC.
...
```

Specifying Size and Row ID for Rowset Arrays

Use the `ROWSET FOR` clause, which is placed immediately after `EXEC SQL` and before the statement that uses rowset arrays, to restrict the size of rowsets and to address the rowset by using a row identifier. The size must be less than or equal to the actual allocated size of the rowset. The `ROWSET FOR` clause is not supported with a cursor declaration. However, you can specify size and row ID in a cursor declaration by

using the rowset-derived table syntax presented in [Selecting From Rowset-Derived Tables With a Cursor](#) on page 7-36.

To specify a ROWSET FOR clause for a DML statement that uses rowsets directly, use:

```
EXEC SQL
ROWSET FOR [size-and-index]
SQL-statement;

size-and-index is:
  INPUT SIZE rowset-size-in
  OUTPUT SIZE rowset-size-out
  KEY BY row-id
  INPUT SIZE rowset-size-in, OUTPUT SIZE rowset-size-out
  INPUT SIZE rowset-size-in, KEY BY row-id
  OUTPUT SIZE rowset-size-out, KEY BY row-id
  INPUT SIZE rowset-size-in, OUTPUT SIZE rowset-size-out, KEY
    BY row-id
```

INPUT SIZE rowset-size-in

restricts the size of the input rowset to the specified size, which must be less than or equal to the allocated size for the rowset. The size is an integer literal (exact numeric literal) or a host variable whose type is either unsigned short, signed short, unsigned long, or signed long in C and their corresponding equivalents in COBOL. By default, if the size is not specified, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program.

OUTPUT SIZE rowset-size-out

restricts the size of the output rowset to the specified size, which must be less than or equal to the allocated size for the rowset. The size is an integer literal (exact numeric literal) or a host variable whose type is signed long in C and its corresponding equivalent in COBOL. By default, if the size is not specified, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section of the embedded SQL program. This option is not supported in a cursor declaration. OUTPUT SIZE works only with SELECT ... INTO type statements.

KEY BY row-id

is a zero-based index that identifies each row in the result set of a SELECT or FETCH statement with the particular search-condition in the WHERE clause that caused the row to be part of the result set. For example, if the *row-id* value for a certain row in the result set is 0 (zero), this row matches the search-condition in the first element of the host variable arrays (array index 0 in C, array index 1 in COBOL) in the WHERE clause.

SQL-statement

is any embedded DML statement that uses rowsets directly.

Limiting the Size of the Input Rowset

When you are inserting rows from a rowset, you must limit the input size to only the rows that have been populated with data.

Example

This example inserts multiple rows (JOBCODE and JOBDESC columns) from host variable arrays into the JOB table. The input size is limited by the FOR INPUT SIZE clause to five rows:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[10] unsigned NUMERIC (4)   hva_jobcode;
    ROWSET[10] VARCHAR                hva_jobdesc[19];
...
EXEC SQL END DECLARE SECTION;

...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
strcpy(hva_jobdesc[0], "PROJECT MANAGER");
hva_jobcode[1] = 200;
strcpy(hva_jobdesc[1], "PROGRAM MANAGER");
hva_jobcode[2] = 300;
strcpy(hva_jobdesc[2], "QUALITY SUPERVISOR");
hva_jobcode[3] = 400;
strcpy(hva_jobdesc[3], "TECHNICAL OFFICER");
hva_jobcode[4] = 500;
strcpy(hva_jobdesc[4], "EXECUTIVE OFFICER");
...
EXEC SQL ROWSET FOR INPUT SIZE 5
    INSERT INTO persnl.job (jobcode, jobdesc)
        VALUES (:hva_jobcode, :hva_jobdesc);
...
```

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[10] hvajobcode   pic 9(4) comp.
    02 ROWSET[10] hvajobdesc   pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

...
***** Populate the rowset in some way *****
Move 100 TO hvajobcode (1)
Move "PROJECT MANAGER" TO hvajobdesc(1)
Move 200 TO hvajobcode (2)
Move "PROGRAM MANAGER" TO hvajobdesc(2)
Move 300 TO hvajobcode (3)
Move "QUALITY SUPERVISOR" TO hvajobdesc(3)
Move 400 TO hvajobcode (4)
Move "TECHNICAL OFFICER" TO hvajobdesc(4)
Move 500 TO hvajobcode (5)
Move "EXECUTIVE OFFICER" TO hvajobdesc(5)
EXEC SQL ROWSET FOR INPUT SIZE 5
    INSERT INTO job (jobcode, jobdesc)
```

```

    values (:hva_jobcode, :hva_jobdesc)
END-EXEC.
...

```

Limiting the Size of the Input Rowset When Declaring a Cursor

When you are declaring a cursor to fetch rows from the database, you can limit the size of the input rowset. Use this general syntax in the cursor declaration when you limit rowset size.

```

DECLARE { cursor-name | ext-cursor-name }
        CURSOR FOR { rowset-clause | ext-statement-name }

rowset-clause is:
    ROWSET FOR [ INPUT SIZE rowset-size-in ]
               [ KEY BY index-identifier ]
               [ INPUT SIZE rowset-size-in,
                 KEY BY index-identifier ]
               <sql-statement><sql-terminator>

```

For information on all syntax elements of DECLARE CURSOR, see the *SQL/MX Reference Manual*.

Example

C

```

EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET[10] unsigned NUMERIC (4) hva_jobcode;
ROWSET[10] VARCHAR hva_jobdesc[19];
NUMERIC(4) input_size;

...
EXEC SQL END DECLARE SECTION;
...

EXEC SQL DECLARE C1 CURSOR FOR
ROWSET FOR INPUT SIZE :input_size
SELECT jobdesc FROM persnl.job
WHERE jobcode = :hva_jobcode;

input_size = 3
/* Populate first 3 rows of input rowset. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 300;

EXEC SQL OPEN C1;

EXEC SQL FETCH C1 INTO :hva_jobdesc;
/* Only rows with jobcode 100, 200 or 300 will be returned */

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate pic x(5).
01 rs.

```

COBOL

```

02 ROWSET[10] hva_jobcode pic 9(4) comp.
02 ROWSET[10] hva_jobdesc pic x(18).
02 input_size pic 9(4) comp
EXEC SQL END DECLARE SECTION END-EXEC.
...

EXEC SQL DECLARE C1 CURSOR FOR
ROWSET FOR INPUT SIZE :input_size
SELECT jobdesc FROM persnl.job
WHERE jobcode = :hva_jobcode END-EXEC.

Move 3 TO input_size
***** Populate first 3 rows of input rowset. *****
Move 100 TO hva_jobcode(1)
Move 200 TO hva_jobcode(2)
Move 300 TO hva_jobcode(3)

EXEC SQL OPEN C1 END-EXEC.

EXEC SQL FETCH C1 INTO :hva_jobdesc END-EXEC.
*** Only rows with jobcode 100, 200 or 300 will be returned ***

```

Limiting the Size of the Output Rowset

When you select rows into a rowset, you can limit the size of the output rowset only with the ROWSET FOR statement (that is, not in a cursor declaration) and for static rowsets.

Example

This example retrieves multiple rows (JOBCODE and JOBDESC columns) from the JOB table into host variable arrays. The output size is limited by the FOR OUTPUT SIZE clause to five rows:

C

```

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ROWSET[10] unsigned NUMERIC (4)   hva_jobcode;
  ROWSET[10] VARCHAR                hva_jobdesc[19];
  long outputsize;
...
EXEC SQL END DECLARE SECTION;
...
outputsize=5;
EXEC SQL ROWSET FOR OUTPUT SIZE :outputsize
  SELECT jobcode, jobdesc
  INTO:hva_jobcode, :hva_jobdesc
  FROM persnl.job;
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate pic x(5).
01 rs.
   02 ROWSET[10] hvajobcode pic 9(4) comp.
   02 ROWSET[10] hvajobdesc pic x(18).
01 outputsize pic s9(9) comp.

```

```

EXEC SQL END DECLARE SECTION END-EXEC.
...
Move 5 to outputsize
EXEC SQL ROWSET FOR OUTPUT SIZE :outputsize
  SELECT jobcode, jobdesc
  INTO :hvajobcode, :hvajobdesc
  FROM job
END-EXEC.
...

```

Using the Index Identifier

Use the index (or row) identifier to indicate which row of the input rowset array in the WHERE clause caused a row to be part of the output rowset array. When you use a zero-based index, the values of the index identifier range from 0 to $n-1$, where n is the number of elements in the WHERE clause rowset array. This strategy might not work as well for a COBOL application where host language array indexing starts from 1. In the next COBOL example, you can add 1 to all the index identifier values by using an arithmetic expression in the select list.

To use the index identifier, you can declare a host variable array, whose size is at least as large as the other output host variable arrays in the SQL statement in the DECLARE section. You can then use a SELECT (or FETCH) operation into this host variable array after including the index identifier (`row_id` in the next example) in the list of columns to be retrieved.

You can also use a cursor declaration to use the index identifier by using the general DECLARE CURSOR syntax shown on page [7-27](#).

Example

This example selects the EMPNUM column of all rows in the EMPLOYEE table where the JOBCODE value is equal to one of the values in the `hva_jobcode` host variable array. The SELECT statement is executed for each matching job code. The row identifier indicates which element of the host variable array selects the corresponding row from the EMPLOYEE table:

C

```

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ROWSET[5] unsigned NUMERIC (4)      hva_jobcode;
  ROWSET [100] unsigned NUMERIC (4)    hva_empnum;
  ROWSET [100] short                   hva_row_id;
  ...
  long numrows;
EXEC SQL END DECLARE SECTION;
  long i;
...
/* Populate the jobcode rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 350;      /* Does not exist. */
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;

```

```

...
EXEC SQL ROWSET FOR KEY BY row_id
  SELECT empnum, row_id
  INTO :hva_empnum,
       :hva_row_id
  FROM persnl.employee
  WHERE jobcode = :hva_jobcode;
...
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
...
for (i = 0; i < numrows; i++) {
  printf("\nEmp Nbr: %hu", hva_empnum[i]);
  printf("\nRow Id: %hu", hva_row_id[i]);
}
...
...
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
   02 ROWSET[5] hvajobcode  pic 9(4) comp.
   02 ROWSET[100] hvaempnum pic 9(4) comp.
   02 ROWSET[100] hvarowid  pic s9(4) comp.
01 numrows                 pic 9(9) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                       pic s9(4) comp.
...
***** Populate the rowset in some way *****
Move 100 TO hvajobcode(1)
Move 200 TO hvajobcode(2)
***3 Does not exist***
Move 350 TO hvajobcode(3)
Move 400 TO hvajobcode(4)
Move 500 TO hvajobcode(5)
EXEC SQL ROWSET FOR KEY BY row_id
  SELECT empnum, row_id+1
  INTO :hvaempnum,
       :hvarowid
  FROM employee
  WHERE jobcode = :hvajobcode END-EXEC.
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT end-exec.
PERFORM VARYING i FROM 1 BY 1 UNTIL i < numrows
  display "Emp Nbr:  " hvaempnum(i)
  display "Row Id:  " hvarowid(i)
end-perform.
...

```

The output for this example has 20 rows selected from the EMPLOYEE table:

- Eleven rows with jobcode equal to 100 and row identifier value 0 for C and 1 for COBOL
- One row with jobcode equal to 200 and row identifier value 1 for C and 2 for COBOL
- Five rows with jobcode equal to 400 and row identifier value 3 for C and 4 for COBOL

- Three rows with jobcode equal to 500 and row identifier value 4 for C and 5 for COBOL

The jobcode equal to 350 does not exist in the sample database. As a result, the row identifier equal to 2 for C and 3 for COBOL does not occur in the output of the program.

The row identifier values in the COBOL example are greater, by a value of 1, than their corresponding values in the C example. This difference occurs because the SQL query is different in the two examples. For SQL, the row identifier column is a zero-based index. For the convenience of COBOL applications, the COBOL example modifies an SQL query to output values of the row identifier column as a one-based index.

Note. Many of the examples in this manual use the NonStop SQL/MX Release 2.x sample database, which uses SQL/MX format tables. To install the sample database, you must have a license for the use of SQL/MX DDL statements. To acquire the license, you must purchase product T0394. If you did not purchase T0394 and you try to install the sample database, an error message informs you that the system is not licensed.

Use the index identifier to obtain a count of how many rows are selected due to each condition in the WHERE clause input rowset array. See the next example, which uses the same table and input host variables as in the previous example:

C

```
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
ROWSET[5] unsigned NUMERIC (4)      hva_jobcode;
ROWSET [100] short                  hva_row_count;
ROWSET [100] short                  hva_row_id;
...
long numrows;
EXEC SQL END DECLARE SECTION;
long i;

...
/* Populate the jobcode rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 350;      /* Does not exist. */
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;
...
EXEC SQL ROWSET FOR KEY BY row_id
SELECT row_id, COUNT(*)
INTO :hva_row_id, :hva_row_count
FROM persnl.employee
WHERE jobcode = :hva_jobcode
GROUP BY row_id ;
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT;
...
for (i = 0; i < numrows; i++) {
    printf("\nRow Id: %hu", hva_row_id[i]);
    printf("\nRow Count: %hu", hva_row_count[i]);
```

```

}
...
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[5] hvajobcode    pic 9(4) comp.
    02 ROWSET[100] hvarowcount pic 9 comp.
    02 ROWSET[100] hvarowid    pic s9(4) comp.
01 numrows                pic 9(9) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                      pic s9(4) comp.
...
***** Populate the rowset in some way *****
Move 100 TO hvajobcode(1)
Move 200 TO hvajobcode(2)
****3 Does not exist****
Move 350 TO hvajobcode(3)
Move 400 TO hvajobcode(4)
Move 500 TO hvajobcode(5)
EXEC SQL ROWSET FOR KEY BY row_id
    SELECT row_id, COUNT(*)
    INTO :hvarowid, :hvarowcount
    FROM employee
    WHERE jobcode = :hvajobcode
    GROUP BY row_id END-EXEC.
EXEC SQL GET DIAGNOSTICS :numrows = ROW_COUNT end-exec.
PERFORM VARYING i FROM 1 BY 1 UNTIL i > numrows
    display "Row Id: " hvarowid(i)
    display "Row Count: " hvarowcount(i)
end-perform.
...

```

Specifying Rowset-Derived Tables

Use a rowset-derived table to manipulate rowsets like other tables in SQL statements. A rowset-derived table is similar to an in-memory table and you can use it, followed by its rowset table correlation, anywhere a table name is specified in a DML statement by using the syntax. (This table correlation clause is required.)

```

ROWSET [rowset-size] (:array-name [, :array-name]...)
    [KEY BY row-id]
    [AS correlation (column [,column]...)]

```

rowset-size

restricts the size of the rowset-derived table to the specified size, which must be less than or equal to the allocated size for the rowset. The size, if specified, immediately follows the ROWSET keyword. The size is an unsigned integer or a host variable whose value is an unsigned integer. By default, if the size is not specified, NonStop SQL/MX uses the allocated rowset size specified in the SQL Declare Section.

`:array-name [,:array-name]...`

specifies a set of host variable arrays. Each *array-name* can be used like a column in the rowset-derived table. Each *array-name* can be any valid host language identifier with a data type that corresponds to an SQL data type. Precede each *array-name* with a colon (:) within an SQL statement.

`KEY BY row-id`

optionally identifies each tuple or row processed in the rowset-derived table during the evaluation of the SQL statement. The *row-id*, if specified, must be the last variable specification in the derived column list.

`[AS] correlation`

is the correlation name of the table reference and can be any SQL identifier.

`column [,column]...`

specifies the list of derived columns of the rowset-derived table that corresponds one-to-one to the list `:array-name [,:array-name]...` of array names, with the exception that the last *column* in the list must be the *row-id*, if specified.

Using Rowset-Derived Tables in DML Statements

Use rowset-derived tables in DML statements:

Technique

Description

[Selecting From Rowset-Derived Tables](#)

Single execution with an input rowset instead of multiple executions with individual input values.

[Inserting Rows From Rowset-Derived Tables](#)

Multiple rows are inserted by using a query that retrieves values from a rowset-derived table.

[Updating Rows by Using Rowset-Derived Tables](#)

Multiple logical executions of an UPDATE statement are performed by using a subquery in the WHERE clause.

[Deleting Rows by Using Rowset-Derived Tables](#)

Multiple logical executions of the DELETE statement are performed by using a subquery in the WHERE clause.

Selecting From Rowset-Derived Tables

Use a rowset as input in a SELECT statement to improve performance. A single execution of the statement is performed with the input rowset instead of multiple executions of an equivalent statement with successive individual values for input.

A SELECT statement that contains a rowset-derived table within the FROM clause handles its input data as a join of the other table references with the rowset-derived table.

Example

In this example, the ODETAIL table is joined with a rowset using a rowset-derived table. The program counts the number of elements in common between the part number values in ODETAIL and the values in the rowset, which is composed of an array of part numbers:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET [5] unsigned NUMERIC (4)   hva_partnum;
    ROWSET [5] unsigned NUMERIC (4)   hva_od_partnum;
    ROWSET [5] short                  hva_partnum_count;
...
EXEC SQL END DECLARE SECTION;
    long i;

...
/* Populate the rowset in some way. */
hva_partnum[0] = 244;
hva_partnum[1] = 2001;
hva_partnum[2] = 2403;
hva_partnum[3] = 5103;
hva_partnum[4] = 6301;
...
EXEC SQL
    SELECT od.partnum, COUNT (*)
    INTO :hva_od_partnum, :hva_partnum_count
    FROM sales.odetail od,
         ROWSET(:hva_partnum) AS rs(partnum)
    WHERE od.partnum = rs.partnum
    GROUP BY od.partnum;

...
/* Process the counts in some way. */
for (i = 0; i < 5; i++) {
    printf("\nPart Nbr: %hu", hva_od_partnum[i]);
    printf("\nCount: %hu", hva_partnum_count[i]);
}
...
```

COBOL

```
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[5] hvapartnum    pic 9(4) comp.
    02 ROWSET[5] hvaodpartnum  pic 9(4) comp.
    02 ROWSET[5] hvapartnumcount pic s9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
01 i                        pic s9(4) comp.
...
***** Populate the rowset in some way *****
Move 244 TO hvapartnum(1)
Move 2001 TO hvapartnum(2)
```

```

Move 2403 TO hvapartnum(3)
Move 5103 TO hvapartnum(4)
Move 6301 TO hvapartnum(5)
EXEC SQL
    SELECT od.partnum, COUNT(*)
    INTO :hvaodpartnum, :hvapartnumcount
    FROM odetail od,
         ROWSET(:hvapartnum) AS rs(partnum)
    WHERE od.partnum = rs.partnum
    GROUP BY od.partnum END-EXEC.
****Process the counts in some way****
PERFORM VARYING i FROM 1 BY 1 UNTIL i < 5
    display "Part Nbr:  " hvaodpartnum(i)
    display "COUNT:  " hvapartnumcount(i)
END-PERFORM.
...

```

Example

This example selects the element of the rowset-derived table that is indexed by the number 4:

C

```

EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET [5] unsigned NUMERIC (4)  hva_partnum;
    unsigned NUMERIC (4)              row_id_partnum;
...
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_partnum[0] = 244;
hva_partnum[1] = 2001;
hva_partnum[2] = 2403;
hva_partnum[3] = 5103;
hva_partnum[4] = 6301;
...
EXEC SQL
    SELECT partnum INTO :row_id_partnum
    FROM ROWSET(:hva_partnum)
         KEY BY row_id AS rs(partnum, row_id)
    WHERE row_id = 4;
...
/* Process the selected element of the table in some way. */
printf("\nPart Nbr: %hu", row_id_partnum);
...

```

In this example, the selected element, whose row identifier is equal to the number 4, is the part number 6301:

COBOL

```

...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rowidpartnum             pic 9(4) comp.
01 rs.
    02 ROWSET[5] hvapartnum    pic 9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.

```

```

...
***** Populate the rowset in some way *****
Move 244 TO hvapartnum(1)
Move 2001 TO hvapartnum(2)
Move 2403 TO hvapartnum(3)
Move 5103 TO hvapartnum(4)
Move 6301 TO hvapartnum(5)
EXEC SQL
    SELECT partnum INTO :rowidpartnum
    FROM ROWSET(:hvapartnum)
        KEY BY row_id AS rs(partnum, row_id)
    WHERE row_id+1 = 5 END-EXEC.
***** Process the selected element of the table in some way ****
Display "Part Nbr  " rowidpartnum.
...

```

Selecting From Rowset-Derived Tables With a Cursor

Declare cursors with rowset-derived tables and use them to fetch rows from the database. This strategy has an advantage over direct use of rowsets arrays with cursors because you can specify *rowset-size* and *row-id* for cursors that use rowset-derived tables, but you cannot specify them for cursors that use rowset arrays directly.

Example

In this example, the ODETAIL table is joined with a rowset using a rowset-derived table. The program uses a cursor to fetch all rows whose order number values are specified in the input rowset, which contains an array of order numbers. The number of valid elements in the input array is specified using a host variable. You can use the index identifier *rowid* to determine which input condition cause a specific row to be output. The *rowid* array is empty before execution of the query. After execution, because *rowid* is a zero-based index, it contains values that range from 0 to the number of valid input conditions minus 1. This strategy might not be optimal for a COBOL application where the array indexing starts from 1. Therefore, for COBOL programs, use the solution in: [Using the Index Identifier](#) on page 7-29. A value *j* in the *rowid* array for a particular row indicates that the row was fetched from the table during execution of input condition number *j* (calculated with a zero-based array indexing).

C

```

EXEC SQL BEGIN DECLARE SECTION;
    long                               SQLCODE;
    ROWSET [5] unsigned NUMERIC (4)   hva_ordernum;
    ROWSET [5] unsigned NUMERIC (4)   hva_od_partnum;
    ROWSET [5] short                   rowid;
    short                               num_inputvalues;
...
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_ordernum[0] = 244;
hva_ordernum[1] = 2001;

```

```

hva_ordernum[2] = 2403;
hva_ordernum[3] = 5103;
...
/* Specify number of valid input values */

num_inputvalues = 4;

/* Declare cursor C1 for select operation */
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT od.partnum, rs.rowid
FROM sales.odetail od,
ROWSET :num_inputvalues(:hva_ordernum)
KEY BY rowid
AS rs(ordernum, rowid)
WHERE od.ordernum = rs.ordernum;

EXEC SQL OPEN C1;

/* Fetch rows from table */
while (SQLCODE == 0) {
EXEC SQL FETCH C1 INTO :hva_od_partnum, :rowid;
/* Process the output rows in some way. */
}
EXEC SQL CLOSE C1;
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLCODE                                pic s9(9) comp.
01 rs.
   02 ROWSET[5] hvaordernum              pic 9(4) comp.
   02 ROWSET[5] hvaodpartnum             pic 9(4) comp.
   02 ROWSET[5] rowid                    pic s9(4) comp.
01 numinputvalues                        pic 9 comp.
EXEC SQL END DECLARE SECTION END-EXEC.

...
***** Populate the rowset in some way *****
Move 244 TO hvaordernum(1)
Move 2001 TO hvaordernum(2)
Move 2403 TO hvaordernum(3)
Move 5103 TO hvaordernum(4)

..
***** Specify number of valid input values *****

Move 4 TO numinputvalues
***** Declare cursor C1 for select operation *****

EXEC SQL
DECLARE C1 CURSOR FOR
SELECT od.partnum, rs.rowid
FROM sales.odetail od,
ROWSET :numinputvalues(:hvaordernum)
KEY BY rowid
AS rs(ordernum, rowid)
WHERE od.ordernum = rs.ordernum

```

```

END-EXEC
EXEC SQL OPEN C1 END-EXEC

**** fetch rows from table ****
perform until sqlcode not = 0
EXEC SQL FETCH C1 INTO :hvaodpartnum, :rowid END-EXEC
**** Process the output rows in some way ****
end-perform
EXEC SQL CLOSE C1 END-EXEC
...

```

Inserting Rows From Rowset-Derived Tables

Use the INSERT statement and rowset-derived tables to insert multiple rows into a table from a query that retrieves from the derived table.

Use this general syntax:

```

INSERT INTO table-name (column [,column]...)
  SELECT column [,column]...
  FROM ROWSET [rowset-size] (:array-name [, :array-name]...)
    [AS] correlation (column [,column]...)

```

For complete syntax, see the INSERT statement in the *SQL/MX Reference Manual*.

Example

This example inserts multiple rows (JOBCODE and JOBDESC columns) selected from a rowset-derived table:

C

```

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ROWSET[5] unsigned NUMERIC (4)   hva_jobcode;
  ROWSET[5] VARCHAR                 hva_jobdesc[19];
...
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
strcpy(hva_jobdesc[0], "PROJECT MANAGER");
hva_jobcode[1] = 200;
strcpy(hva_jobdesc[1], "PROGRAM MANAGER");
hva_jobcode[2] = 300;
strcpy(hva_jobdesc[2], "QUALITY SUPERVISOR");
hva_jobcode[3] = 400;
strcpy(hva_jobdesc[3], "TECHNICAL OFFICER");
hva_jobcode[4] = 500;
strcpy(hva_jobdesc[4], "EXECUTIVE OFFICER");
...
EXEC SQL INSERT INTO persnl.job
      SELECT jobcode, jobdesc
      FROM ROWSET(:hva_jobcode, :hva_jobdesc)

```

```

AS rs(jobcode, jobdesc);
...
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[5] hvajobcode  pic 9(4) comp.
    02 ROWSET[5] hvajobdesc  pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.
**** Populate the rowset in some way ****
Move 100 TO hvajobcode (1)
Move "PROJECT MANAGER" TO hvajobdesc(1)
Move 200 TO hvajobcode (2)
Move "PROGRAM MANAGER" TO hvajobdesc(2)
Move 300 TO hvajobcode (3)
Move "QUALITY SUPERVISOR" TO hvajobdesc(3)
Move 400 TO hvajobcode (4)
Move "TECHNICAL OFFICER" TO hvajobdesc(4)
Move 500 TO hvajobcode (5)
Move "EXECUTIVE OFFICER" TO hvajobdesc(5)
EXEC SQL INSERT INTO job
    SELECT jobcode, jobdesc
    FROM ROWSET(:hvajobcode, :hvajobdesc)
    AS rs(jobcode, jobdesc) END-EXEC.
...

```

Limiting the Size of a Rowset-Derived Table

When you are inserting rows from a rowset-derived table, you must limit the input size to only the rows that have been populated with data.

Example

This example inserts multiple rows into the JOB table. The input size is limited by the size of the rowset-derived table:

```

C EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ROWSET[10] unsigned NUMERIC (4)  hva_jobcode;
  ROWSET[10] VARCHAR                hva_jobdesc[19];
...
EXEC SQL END DECLARE SECTION;
...
/* Populate the first five rows in some way. */
...
EXEC SQL INSERT INTO persnl.job
    SELECT jobcode, jobdesc
    FROM ROWSET 5 (:hva_jobcode, :hva_jobdesc)
    AS rs(jobcode, jobdesc);
...
...
COBOL EXEC SQL BEGIN DECLARE SECTION END-EXEC.

```

```

01 sqlstate                      pic x(5).
01 rs.
   02 ROWSET[10] hvajobcode      pic 9(4) comp.
   02 ROWSET[10] hvajobdesc      pic x(18).
EXEC SQL END DECLARE SECTION END-EXEC.

...
***** Populate the first five rows in some way *****
EXEC SQL INSERT INTO persnl.job
      SELECT jobcode, jobdesc
      FROM ROWSET 5(:hvajobcode, :hvajobdesc)
      AS rs(jobcode, jobdesc) END-EXEC.

...

```

Inserting Null

Use an indicator host variable array in a rowset-derived table to insert multiple rows of data with a null value for one of the columns in some of the rows.

Example

This example inserts 100 rows into the EMPLOYEE table and sets the SALARY column to null for the first 50 rows by using an indicator host variable array in a rowset-derived table. The null indicator is -1. For the remaining 50 rows, the SALARY column is set to nonnull values. The indicator host variable array must contain the value 0 (zero) for these rows:

C

```

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ROWSET [100] unsigned NUMERIC (4)   hva_empnum;
  ROWSET [100] char                  hva_first_name[16];
  ROWSET [100] char                  hva_last_name[21];
  ROWSET [100] unsigned NUMERIC (4)   hva_deptnum;
  ROWSET [100] unsigned NUMERIC (4)   hva_jobcode;
  ROWSET [100] unsigned NUMERIC (4)   hva_salary;
  ROWSET [100] short                 hva_salary_indicator;
...
EXEC SQL END DECLARE SECTION;

...
/* Populate the host variable arrays in some way. */
...
/* Store -1 in the indicator array for the first 50 input
values. */
for (i = 0; i < 50; i++) hva_salary_indicator[i] = -1;
/* Store 0 in the indicator array for the next 50 input values.
It is assumed that there are valid values for salary in the
hva_salary rowset array from element no. 50 up to element no. 99
*/
for (i = 50; i < 100; i++) hva_salary_indicator[i] = 0;
EXEC SQL
  INSERT INTO persnl.employee
  SELECT (empnum,first_name,last_name,deptnum,jobcode,salary)
  FROM
  ROWSET(:hva_empnum,:hva_first_name,
        :hva_last_name,:hva_deptnum,:hva_jobcode,

```



```

        :hva_salary INDICATOR :hva_salary_indicator)
    AS rs(empnum,first_name,last_name,deptnum,jobcode,salary);
...

```

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE                                pic x(5).
01 rs.
    02 ROWSET[100] hvaempnum                pic 9(4) comp.
    02 ROWSET[100] hvafirstname            pic x(15).
    02 ROWSET[100] hvalastname             pic x(20).
    02 ROWSET[100] hvadeptnum              pic 9(4) comp.
    02 ROWSET[100] hvajobcode              pic 9(4) comp.
    02 ROWSET[100] hvasalary               pic 9(4) comp.
    02 ROWSET[100] hvasalaryindicator      pic 9(5).
EXEC SQL END DECLARE SECTION END-EXEC.

...
**** Populate the host variables arrays in some way ****
**** Store -1 in the indicator array for the first 50 ****
**** input values. ****
PERFORM VARYING i FROM 1 BY 1 UNTIL i = 50
    Move -1 to hvasalaryindicator(i)
end-perform.

**** Store 0 in the indicator array for the next 50 ****
**** input values. It is assumed that there are valid ****
**** values for salary in the hvasalary rowset array ****
**** from element no. 51 upto element no. 100 ****
PERFORM VARYING i FROM 51 BY 1 UNTIL i = 100
    Move 0 to hvasalaryindicator(i)
end-perform.

EXEC SQL
    INSERT INTO employee
    SELECT empnum, first_name, last_name, deptnum,
           jobcode, salary
    FROM
    ROWSET(:hvaempnum, :hvafirstname, :hvalastname,
           :hvadeptnum, :hvajobcode,
           :hvasalary INDICATOR :hvasalaryindicator)
    AS rs(empnum, first_name, last_name, deptnum,
           jobcode, salary)
END-EXEC.
...

```

Updating Rows by Using Rowset-Derived Tables

Use a rowset-derived table in an UPDATE statement to indicate which rows are to be updated from the database table. In this case, the values of the rowset are generated from a subquery placed in the WHERE clause of the UPDATE statement.

Example

This example updates the SALARY column of all rows in the EMPLOYEE table where the jobcode value is equal to one of the values in the hva_jobcode host variable array. The UPDATE statement is executed for each matching job code:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[5] unsigned NUMERIC (4)  hva_jobcode;
...
EXEC SQL END DECLARE SECTION;
...
/* Input the salary increment. */
...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 300;
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;
...
EXEC SQL
    UPDATE persnl.employee
    SET salary = salary * :hv_inc
    WHERE EXISTS
        (SELECT * FROM ROWSET(:hva_jobcode) AS rs(jobcode)
         WHERE jobcode = rs.jobcode);
...
```

The number of updated rows is stored in the ROW_COUNT field of the statement information in the diagnostics area. Retrieve the value in the ROW_COUNT field by using the GET DIAGNOSTICS statement.

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 sqlstate                pic x(5).
    01 hvainc                  pic 9(4).
    01 rs.
        02 ROWSET[5] hvajobcode    pic 9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
...
**** Input the salary increment ****
**** Populate the rowset in some way ****
Move 100 TO  hvajobcode(1).
Move 200 TO  hvajobcode(2).
Move 300 TO  hvajobcode(3).
Move 400 TO  hvajobcode(4).
Move 500 TO  hvajobcode(5).
EXEC SQL
    UPDATE employee
    SET salary = salary * :hvainc
    WHERE EXISTS
        (SELECT * FROM ROWSET(:hvajobcode) AS rs(jobcode)
         WHERE jobcode = rs.jobcode) END-EXEC.
...
```

Deleting Rows by Using Rowset-Derived Tables

Use a rowset-derived table in a DELETE statement to indicate which rows are to be deleted from the database table. In this case, the values of the rowset are generated from a subquery placed in the WHERE clause of the DELETE statement.

Example

This example deletes all rows from the JOB table specified by the `hva_jobcode` host variable array:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ROWSET[5] unsigned NUMERIC (4)  hva_jobcode;
...
EXEC SQL END DECLARE SECTION;
...
/* Populate the rowset in some way. */
hva_jobcode[0] = 100;
hva_jobcode[1] = 200;
hva_jobcode[2] = 300;
hva_jobcode[3] = 400;
hva_jobcode[4] = 500;
...
EXEC SQL
    DELETE FROM persnl.job
    WHERE jobcode IN
        (SELECT jobcode FROM ROWSET(:hva_jobcode) AS rs(jobcode));
...
```

The number of deleted rows is stored in the ROW_COUNT field of the statement information in the diagnostics area. You can retrieve the value in the ROW_COUNT field by using the GET DIAGNOSTICS statement.

COBOL

```
...
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate                pic x(5).
01 rs.
    02 ROWSET[5] hvajobcode  pic 9(4) comp.
EXEC SQL END DECLARE SECTION END-EXEC.
...
***** Populate the rowset in some way *****
Move 100 TO hvajobcode(1)
Move 200 TO hvajobcode(2)
Move 300 TO hvajobcode(3)
Move 400 TO hvajobcode(4)
Move 500 TO hvajobcode(5)

EXEC SQL
    DELETE FROM job
    WHERE jobcode IN
        (SELECT jobcode FROM ROWSET(:hvajobcode)
         AS rs(jobcode))
END-EXEC.
...
```


Name Resolution, Similarity Checks, and Automatic Recompilation

This section covers the following:

- [Name Resolution](#) on page 8-1
- [Similarity Checks and Automatic Recompilation](#) on page 8-9

Name Resolution

In a C, C++, and COBOL program, you can use SQL/MX statements to query both SQL/MP and SQL/MX database objects. This subsection explains how to refer to SQL/MP and SQL/MX database objects in an C, C++, and COBOL source file and how the object names are resolved during SQL compilation and run time. See these topics:

- [Table and View Name References](#) on page 8-1
- [Precedence of Object Name Qualification](#) on page 8-5
- [Compile-Time Name Resolution for SQL/MP Objects](#) on page 8-6
- [Late Name Resolution](#) on page 8-6
- [Distributed Database Considerations](#) on page 8-8
- [RDF Considerations](#) on page 8-8

Table and View Name References

When you write a static SQL statement in an embedded SQL program, you have several choices of how to refer to tables or views:

- [ANSI Names for SQL/MX Objects](#) on page 8-2
- [Guardian Names for SQL/MP Objects](#) on page 8-2
- [SQL/MP Aliases for SQL/MP Objects](#) on page 8-2
- [DEFINE Names for SQL/MP Objects](#) on page 8-3
- [PROTOTYPE Host Variables For SQL/MP and SQL/MX Objects](#) on page 8-4

You can fully or partially qualify ANSI logical names, Guardian physical names, and SQL/MP aliases. For more information, see [Precedence of Object Name Qualification](#) on page 8-5.

Your choice of how to refer to a table or view directly influences how tightly a physical table is bound to a table or view name in the SQL statement during SQL compilation.

ANSI Names for SQL/MX Objects

You can use only an ANSI logical name for SQL/MX tables or views in an SQL statement:

```
EXEC SQL
  SELECT jobcode, jobdesc
  INTO :hv_jobcode, hv_jobdesc
  FROM newyork.persnl.job
  WHERE jobcode = :hv_this_jobcode;
```

When you hard code a table name in the ANSI logical format, the logical name is tightly bound to a physical table when the SQL statement is compiled. To make the statement refer to some other physical table, you need to edit the source code and recompile the module.

For the syntax of this type of database object name, see the *SQL/MX Reference Manual*.

Guardian Names for SQL/MP Objects

You can use the Guardian physical name for SQL/MP tables or views in an SQL statement:

```
EXEC SQL
  SELECT jobcode, jobdesc
  INTO :hv_jobcode, hv_jobdesc
  FROM \ny.$data01.persnl.job
  WHERE jobcode = :hv_this_jobcode;
```

When you hard code a table name in the Guardian format, the physical name is tightly bound to the SQL statement when the SQL statement is compiled. To make the statement refer to some other physical table, you must edit the source code and recompile the module.

For the syntax of this type of database object name, see the *SQL/MX Reference Manual*.

SQL/MP Aliases for SQL/MP Objects

To use logical names for SQL/MP tables or views, create an SQL/MP alias that maps to a Guardian physical name and refer to the SQL/MP alias in the SQL statement.

To create an SQL/MP alias, issue a CREATE SQLMP ALIAS statement in MXCI:

```
CREATE SQLMP ALIAS samdbcat.persnl.employee
  $samdb.persnl.employee;
```

In the source file, refer to the SQL/MP alias as:

```
EXEC SQL DELETE FROM samdbcat.persnl.employee
```

If you embed a CREATE SQLMP ALIAS statement in your embedded SQL program, subsequent references to the SQL/MP alias in the same program cause compilation

errors. To avoid these errors, create SQL/MP aliases separately before compiling the embedded SQL program.

When you code a table name by using an SQL/MP alias, the logical name of the SQL/MP alias is tightly bound to a physical table when the SQL statement is compiled. To make the statement refer to some other physical table, you must alter the SQL/MP alias and then recompile the module. For example, issue this ALTER SQLMP ALIAS statement before recompiling the module.

```
ALTER SQLMP ALIAS samdbcat.persnl.employee  
    $samdb.persnl.newemps;
```

For the syntax of the CREATE SQLMP ALIAS and ALTER SQLMP ALIAS statements, see the *SQL/MX Reference Manual*.

DEFINE Names for SQL/MP Objects

Note. Before using DEFINES in OLTP applications, see [OLT Optimization Considerations for DEFINE Names and PROTOTYPE Host Variables](#) on page 8-5.

You can refer to an SQL/MP table or view with a class MAP DEFINE that resolves to a Guardian physical name:

```
EXEC SQL  
    SELECT jobcode, jobdesc  
    INTO :hv_jobcode, hv_jobdesc  
    FROM =JOB  
    WHERE jobcode = :hv_this_jobcode;
```

When you use a DEFINE to refer to a table or view, you can compile the statement's module with the DEFINE mapped to one table or view name, and then assign the DEFINE a different name and recompile the module. The recompiled statement then refers to the second table or view name for the DEFINE. This practice is called *compile-time name resolution*. For more information, see [Compile-Time Name Resolution for SQL/MP Objects](#) on page 8-6.

The use of DEFINES in SQL statements also enables late name resolution. By using late name resolution, you can compile a statement to use one table and then, without recompiling the statement, process a different table when the statement is executed by using a different value for the statement's DEFINE. For more information, see [Late Name Resolution](#) on page 8-6.

For the syntax of DEFINES, see the *SQL/MX Reference Manual*. To use a DEFINE for a table or view, map it to a Guardian physical file before you compile or execute the statement. For information on how to ensure proper name resolution, see the *SQL/MX Release 3.2 Management Guide*.

PROTOTYPE Host Variables For SQL/MP and SQL/MX Objects

Note. Before using PROTOTYPE host variables in OLTP applications, see [OLT Optimization Considerations for DEFINE Names and PROTOTYPE Host Variables](#) on page 8-5.

When you use a PROTOTYPE host variable to refer to a table or view, the SQL statement is compiled using the definition of the table or view specified in the PROTOTYPE clause. At run time, the plan is executed using the table or view name that you pass to the host variable. Because PROTOTYPE host variables refer dynamically to tables or views, they enable late name resolution. For more information, see [Late Name Resolution](#) on page 8-6.

To specify a PROTOTYPE host variable in place of a table name in an embedded DML statement, use this syntax:

```
:hostvar PROTOTYPE { '[[catalog.]schema.]table'
                      { '\system.'[$volume.]subvolume.]table' }
```

:hostvar

is a host variable that contains the SQL table name at run time. It can be any valid host language identifier with a data type that corresponds to an SQL data type. You must precede *hostvar* with a colon (:) within an SQL statement.

The table name in *hostvar* can be either:

- Three-part logical name for accessing SQL/MP tables (if using MPALIAS) or SQL/MX tables
- Four-part Guardian file name for accessing SQL/MP tables

You must use the same form for both the PROTOTYPE and *hostvar* value. For example, if the value specified in the PROTOTYPE clause is a Guardian file name, the actual name passed in through *hostvar* must also be a Guardian name.

You must initialize *hostvar* with a table name before the statement executes. If a three-part logical name is not fully qualified, NonStop SQL/MX uses default catalog and schema values as described in the *SQL/MX Reference Manual*.

```
PROTOTYPE { '[[catalog.]schema.]table'
            { '\system.'[$volume.]subvolume.]table' }
```

is one of these name types enclosed in single quotation marks ('):

- Three-part logical name of an SQL/MP alias or SQL/MX table of the form *catalog.schema.table*
- Four-part Guardian name of an SQL/MP table of the form *\system.\$volume.subvolume.table*

The SQL/MX compiler uses this name during preprocessing and explicit compilation of the embedded DML statement. The table or view name defined in the PROTOTYPE clause must be visible on the system where the compilation is performed. This table or view name should also be fully qualified so that default catalog and schema names are not used.

PROTOTYPE host variables enable run-time mappings, which differ from class MAP DEFINES for SQL/MP objects. DEFINES for SQL/MP objects enable both compile and run-time mappings. Although PROTOTYPE host variables are mapped at run time, this run-time mapping cannot be used during explicit recompilation of the application using `mxcmp` or `mxCompileUserModule`. The table or view name defined in the PROTOTYPE clause of the program is used during explicit recompilation. However, if your application fails a similarity check during execution and has automatic recompilation turned on, the automatic recompilation uses the host variable value that you specify at run time.

For examples of PROTOTYPE host variables, see [Using PROTOTYPE Host Variables as Table Names](#) on page 5-17.

OLT Optimization Considerations for DEFINE Names and PROTOTYPE Host Variables

If you use DEFINE names or PROTOTYPE host variables in a statement that is optimized for online transaction processing, the resulting plan uses online transaction (OLT) optimization in these paths only:

- PARTITION_ACCESS operator
- DP2 operations

An OLT optimized plan does not use OLT optimization in the ROOT operator if you use DEFINE names or PROTOTYPE host variables in the statement. For information on OLT optimization, see the *SQL/MX Query Guide*.

Precedence of Object Name Qualification

If the object names in an SQL statement are unqualified or partially qualified, the preprocessor determines the qualification—the catalog and schema (or the node, volume, and subvolume)—of the object names based on these settings in order of precedence, from highest to lowest:

1. DECLARE, SET, or CONTROL QUERY DEFAULT statements in the embedded SQL program
2. Default NAMETYPE, CATALOG, SCHEMA, MP_SYSTEM, MP_VOLUME, or MP_SUBVOLUME specified in the SYSTEM_DEFAULTS table
3. User group (catalog) and user name (schema) of the current user if the nametype is ANSI (the default) or the default node, volume, and subvolume specified by the _DEFAULTS DEFINE if the nametype is NSK

For more information on the SYSTEM_DEFAULTS table, see the *SQL/MX Reference Manual*.

Compile-Time Name Resolution for SQL/MP Objects

Compile-time name resolution is an SQL/MX extension you use to compile a module with statements that refer to SQL/MP tables or views with class MAP DEFINES. For each statement, the SQL/MX compiler prepares a plan that is specific to, and optimized for, the physical table referenced in the file attribute of the DEFINE at the time of compilation.

By using compile-time resolution, you can also reinitialize the DEFINES to values that differ from those used when the module was first compiled, and then recompile the module to prepare plans for a different set of tables than the application was originally built to process, without changing the source code of the application.

To use compile-time name resolution to prepare two or more applications from the same source module, where each application processes its own set of tables, consider using the targeting technique of module management. See [Specifying the search locations of the module files](#) on page 17-13. If you do not use this technique, the compiled module file of the second application overwrites the module file of the first application.

Late Name Resolution

Late name or run-time resolution is an SQL/MX extension that enables an embedded SQL program to use class MAP DEFINES and PROTOTYPE host variables in place of table names in DML statements. PROTOTYPE host variables can be used for SQL/MP and SQL/MX objects. During explicit SQL compilation, the SQL/MX compiler uses the tables from the DEFINES or PROTOTYPE (if they are available) to generate a query execution plan for each DML statement. At run time, you can control which table the statement processes by changing the table name in the value of a DEFINE or by passing the table name in the value of a host variable (for PROTOTYPE host variables). You can specify the same table as the one that was originally compiled, or you can specify a different table.

Note. DEFINES are logical names used for SQL/MP objects and cannot be used with SQL/MX objects. For SQL/MX objects, use PROTOTYPE host variables.

Each time a DML statement executes (or a cursor is opened), the SQL/MX executor compares the name of the table for which the plan was compiled against the name taken from either the run-time DEFINE or the host variable. If the run-time DEFINE does not exist, the SQL/MX executor uses the compile-time DEFINE specified in the module. If the compile-time and run-time names do not match, the SQL/MX executor performs a similarity check of the tables to determine if the query execution plan of the DML statement is still operable. If the similarity check fails (or is disabled), the SQL/MX executor, by default, invokes the SQL/MX compiler to automatically recompile the SQL plan. See [Similarity Checks and Automatic Recompilation](#) on page 8-9.

Late Name Resolution for Tables Referred by the View

Unlike SQL/MX tables, the tables referred in a view cannot use PROTOTYPE host variables instead of table names. Late name resolution is a SQL/MX extension that allows table names referred to in a view definition to differ between compile-time and run-time.

To resolve SQL/MX table names accessed by the view, a one-to-one mapping of the tables in compile-time and run-time is performed by the executor. Therefore, the position in which the tables appear in the view text must be the same at compile-time and runtime. Similarity check for the view fails if the positions of the tables or predicates are different, even though both the views are semantically equivalent.

Example

The following example shows how underlying SQL/MX tables in the view are mapped between compile-time and run-time before performing the similarity check for the view:

Consider the following DML statement:

```
SELECT * from :viewname PROTOTYPE 'CAT.SCH.TEMP_VIEW';
```

The view definitions in compile-time and run-time are as follows:

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS  
SELECT T1.X1, T1.Y1, T2.X2  
FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS  
SELECT T1.X1, T1.Y1, T2.X2  
FROM PRDCAT.PRDSCH_ONE.TABLE_ONE T1, PRDCAT.PRDSCH_ONE.TABLE_TWO  
T2;
```

At compile-time, the query is compiled with the view name, CAT.SCH.TEMP_VIEW in the PROTOTYPE clause. At run-time, the query is executed with the view name, PRDCAT.PRDSCH_ONE.PRDTEMP_VIEW passed in the host variable: viewname. Similarity check compares the views, TEMP_VIEW and PRDTEMP_VIEW and maps the underlying tables in the views. Similarity check for the view passes if the conditions described in [Similarity Check Criteria for a View](#) on page 8-13 are fulfilled.

CAT.SCH.TABLE_ONE is mapped to PRDCAT.PRDSCH_ONE.TABLE_ONE and CAT.SCH.TABLE_TWO is mapped to PRDCAT.PRDSCH_ONE.TABLE_TWO. Similarity check verifies if the mapped tables are equivalent in structure.

Distributed Database Considerations

The SQL statements in an embedded SQL program can refer to SQL/MX and SQL/MP database objects on remote nodes.

Remote SQL/MX Objects

To refer to remote SQL/MX database objects in an embedded SQL program, you need not change the database object names in the source code. However, the catalog that contains the SQL/MX objects must be visible (that is, registered) on the local node before you compile and run your embedded SQL program. For information on registering catalogs and managing an SQL/MX distributed database, see the *SQL/MX Release 3.2 Management Guide*.

Remote SQL/MP Objects

To refer to remote SQL/MP objects by Guardian name in an embedded SQL program, you should fully qualify the SQL/MP object name, including the name of the remote node.

If the embedded SQL program uses a class MAP DEFINE or SQL/MP alias name for a remote SQL/MP object, you should specify a fully qualified SQL/MP object name when you add the DEFINE or create the SQL/MP alias. The SQL/MP aliases must be in SQL/MX user catalogs that are visible on the node where the program executes. For information on registering catalogs to make them visible in an SQL/MX distributed database environment, see the *SQL/MX Release 3.2 Management Guide*.

For information on managing an SQL/MP distributed database, see the *SQL/MX Release 3.2 Management Guide*.

Note. NonStop SQL/MX Release 2.x applications cannot query remote SQL/MP objects on a node that has NonStop SQL/MX Release 1.8 installed. To query these remote objects, you must upgrade the node to SQL/MX Release 2.x. For more information, see the *SQL/MX Installation and Management Guide*.

RDF Considerations

The Remote Duplicate Database Facility (RDF) subsystem monitors changes to a production database on a local (primary) system and maintains a copy of the database on a remote (backup) system. RDF stores a backup of the database objects in a different catalog on the backup node than on the primary node. For more information, see the *RDF/IMP, IMPX, and ZLT System Management Manual*.

An embedded SQL application must be able to run on both the primary and backup nodes. Because RDF stores database objects in different catalogs on the primary and backup nodes, applications in an RDF environment should not refer to hard-coded database object names that refer to a specific node or catalog.

To enable easier deployment of your embedded SQL applications in an RDF environment, follow these guidelines:

- [SQL/MP Object Names for an RDF Environment](#) on page 8-9
- [SQL/MX Object Names for an RDF Environment](#) on page 8-9

SQL/MP Object Names for an RDF Environment

When referring to SQL/MP objects in an embedded SQL program, use class MAP DEFINES or SQL/MP aliases. See [DEFINE Names for SQL/MP Objects](#) on page 8-3 and [SQL/MP Aliases for SQL/MP Objects](#) on page 8-2. If you refer to SQL/MP aliases in the program, use partially qualified names by omitting the catalog name. See [SQL/MX Object Names for an RDF Environment](#) on page 8-9.

If you use hard-coded Guardian names, omit the node name from the table name, letting the node default to the node on which the program is preprocessed. See [Guardian Names for SQL/MP Objects](#) on page 8-2.

SQL/MX Object Names for an RDF Environment

When referring to SQL/MX objects in an embedded SQL program, use partially qualified names by omitting the catalog name (for example, *sch.tab* or *tab*). See [ANSI Names for SQL/MX Objects](#) on page 8-2.

Do not embed DECLARE, SET, or CONTROL QUERY DEFAULT statements that qualify the object names. Instead, qualify the object names with the preprocessor options for catalog and schema during preprocessing. For more information, see [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8 and [Running the SQL/MX COBOL Preprocessor](#) on page 16-9.

Similarity Checks and Automatic Recompilation

This subsection explains what causes similarity checks and automatic recompilation to occur and how to control these operations by coding CONTROL QUERY DEFAULT statements in a program. See these topics:

- [Similarity Check](#) on page 8-9
- [Automatic Recompilation](#) on page 8-18
- [Recommended Recompilation Settings for OLTP Programs](#) on page 8-19

Similarity Check

During a similarity check, the SQL/MX executor checks each table or view in a DML statement at run-time to determine if the query execution plan of the statement is still operable. A similarity check is faster than the automatic recompilation of a query execution plan and can reduce the performance costs of automatic recompilation.

If the statement fails the [Similarity Check Criteria](#) (or if the similarity check is disabled), the SQL/MX executor, by default, invokes the SQL/MX compiler to automatically

recompile the SQL plan. For more information, see [Automatic Recompilation](#) on page 8-18.

Causes of a Similarity Check

When enabled, a similarity check occurs for these reasons:

- [Changed DEFINEs or PROTOTYPE Host Variables](#) on page 8-10
- [Failed Timestamp Check](#) on page 8-10

Similarity checks are performed regardless of the method chosen to refer to tables in the source code: hard-coded Guardian or logical names, class map DEFINEs, or PROTOTYPE host variables. However, similarity checks for views are performed only when views are referred using logical names or PROTOTYPE host variables. For more information, see [Name Resolution](#) on page 8-1.

Changed DEFINEs or PROTOTYPE Host Variables

The SQL/MX executor performs a similarity check if a statement that was compiled for one table or view, specified by either a class map DEFINE or a PROTOTYPE host variable, will now be executed to process a different table or view specified by a new DEFINE or host variable value, respectively. For more information, see [Late Name Resolution](#) on page 8-6 and [Late Name Resolution for Tables Referred by the View](#) on page 8-7.

If the value of a DEFINE or a PROTOTYPE host variable changes, the SQL/MX executor initiates a similarity check, comparing the compile-time table or view to the run-time table or view, respectively, to determine if the query execution plan is still operable. See [Similarity Check Criteria](#) on page 8-11 and [Similarity Check Criteria for a View](#) on page 8-13.

Failed Timestamp Check

The SQL/MX executor performs a similarity check if a timestamp check fails, which occurs if the table's or view's redefinition timestamp has changed since the referencing statement was compiled. The SQL/MX executor performs a timestamp check for each table or view referenced in an SQL statement at table or view open time (the first time the table or view is opened). The timestamp check ensures that the current execution plan of an SQL statement uses a valid definition of each table or view.

After the SQL/MX compiler has prepared the query execution plan of a statement, changes to the tables or views that the plan processes can occur. For example, an index can be added or removed, or a partition added. Changes of this nature can render the plan inoperable. These types of changes to a table also change the redefinition timestamp of the table.

Each table or view contains a redefinition timestamp in its file label. At compile time, the timestamp of each table or view accessed by an SQL statement is stored with the compiled plan of the statement in a module. When executing a plan, the SQL/MX executor compares the current timestamp in the table's or view's file label with the compile-time timestamp of the same table or view in the query execution plan. If the

timestamps differ, the SQL/MX executor initiates a similarity check, comparing the compile-time version to the run-time version of the table or view, to determine if the query execution plan is still operable. See [Similarity Check Criteria](#) on page 8-11 and [Similarity Check Criteria for a View](#) on page 8-13.

Controlling the Similarity Check

By default, the similarity check is enabled for all DML statements. To disable the similarity check and force recompilation of an SQL/MX statement when a class MAP DEFINE value or table, or view timestamp changes, use the CONTROL QUERY DEFAULT or CONTROL TABLE statement with the SIMILARITY_CHECK option. For example, this statement disables the similarity check for all tables in subsequent DML statements:

```
CONTROL QUERY DEFAULT SIMILARITY_CHECK 'OFF' ;
```

This statement disables the similarity check for a specific table:

```
CONTROL TABLE samdbcat.persnl.job SIMILARITY_CHECK 'OFF' ;
```

If you need to re-enable the similarity check in the program, use the CONTROL QUERY DEFAULT or CONTROL TABLE statement with the SIMILARITY_CHECK option set to ON. For more information on coding CONTROL statements, see [Using CONTROL Statements](#) on page 2-12 and the *SQL/MX Reference Manual*.

Similarity check for views is controlled at the view creation time by the CONTROL QUERY DEFAULT, DDL_VIEW_SIMILARITY_CHECK. For example, this statement enables the similarity check for all views in subsequent DDL statements:

```
CONTROL QUERY DEFAULT DDL_VIEW_SIMILARITY_CHECK 'ENABLE' ;
```

The CONTROL QUERY DEFAULT only applies in the absence of explicit syntax options for similarity check for views. Similarity check for views can also be enabled using ENABLE SIMILARITY CHECK in a CREATE VIEW or ALTER VIEW statement. For more information, see the *SQL/MX Reference Manual*.

Note. Although the SIMILARITY_CHECK option is enabled by default in NonStop SQL/MX, a DML statement that refers to an SQL/MP table does not undergo a similarity check in NonStop SQL/MX if the SQL/MP table was created with the SIMILARITY CHECK option disabled. To determine if the similarity check is enabled for an SQL/MP table, check the SIMILARITYCHECK column for the table in the TABLES table of the SQL/MP catalog. For more information, see the *SQL/MP Reference Manual*.

Similarity Check Criteria

During a similarity check, the SQL/MX executor compares the compile-time version of a table with its run-time version. For the similarity check to pass:

- Both tables must have the same table type (key-sequenced or entry-sequenced).
- Both tables must be either audited or nonaudited.

- Both tables must have the same number of columns. The similarity check fails if the total number of columns is different, even if the referenced columns in the statement are the same for both tables.
- All corresponding columns in both tables must have the same:
 - Name
 - Data type, length, precision, and scale
 - NULL attribute (Both columns must be either NULL or NOT NULL.) (NOT NULL clauses in SQL/MP and SQL/MX native tables do not disable similarity checks. If a table contains a NOT NULL clause, the SHOWDDL command displays it as part of the CHECK clause. However, it is not handled as a check constraint clause when the similarity information is generated.)
 - DEFAULT clause (For user-specified values, columns must have the same default value. For SQL/MP floating-point columns, the same default value at run time might not always match the compile-time value, causing the similarity check to fail.)
- Both tables must have the same number of key columns. Corresponding key columns must have the same:
 - Position, offset, and column number
 - Column attributes
- If a DML statement uses an index, the run-time table must have an index that has the same attributes as the index that was used by the compile-time table. For an index on the run-time table to be considered similar to the compile-time index:
 - Both must be either unique or nonunique.
 - Both must have the same keytag value (SQL/MP indexes). (See the CREATE INDEX statement in the *SQL/MP Reference Manual* for details about the keytag specification of an index. If the index is created without specifying the keytag, for the indexes of old and new tables to match, they must have been created in the same sequence.)
 - Both must have the same number of key columns.
 - Both must have the same key column attributes.
- For a DML statement using indexes, the indexes must not be partitioned. If the indexes are partitioned, the similarity check fails.
- Both tables must have the same partitioning scheme (hash or range partitioned), the same number of partitions, and the same partitioning keys if the plan does not use OLT optimization.

Note. If the plan uses OLT optimization, the number of partitions does not affect the similarity check. For more information on OLT optimization, see the *SQL/MX Query Guide*.

- Both tables must not have any check constraints or referential integrity constraints.

- The query must not use compound statements.
- ESP parallelism must not have been used.
- The query must not include any SQL/MP views.
- A CALL statement must not refer to a stored procedure in Java (SPJ) that has been dropped and re-created in NonStop SQL/MX.
- An IUD query must not be an embedded update or delete.
- The query must not be an update on primary key or unique index column.

Similarity Check Criteria for a View

During similarity check for a view, the SQL/MX executor compares the view at compile-time and run-time. For the view to pass the similarity check, the following criteria must be satisfied:

- The view text, except for the table name and view name, must remain the same.
- The view must not be a nested view or a view with VALUES clause.
- Similarity check for views must be enabled for all the views referred to in the DML statement.
- The number of view columns must be the same between compile-time and run-time.
- The following attributes must be the same for all the corresponding columns projected by the views:
 - Name
 - Heading
 - Data type
 - Character Set
 - Collation
 - NULL/NOT NULL
 - IDENTITY COLUMN.
- The tables referred by both the views must pass the [Similarity Check Criteria](#) for tables. If any of the criteria is not fulfilled, similarity check for views fails. A warning is issued if the RECOMPILATION_WARNINGS CQD is set to ON.

Examples of similarity check for a view

- In this example, at compile-time, the view refers to two tables, CAT.SCH.TABLE_ONE T1 and CAT.SCH.TABLE_TWO T2. However, at run-time, the view refers to only one table, PRDCAT.PRDSCH_ONE.TABLE_ONE T1.

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT X1, Y1, X2
FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT X1, Y1 FROM PRDCAT.PRDSCH_ONE.TABLE_ONE T1;
```

Thus, view similarity check fails, because of the difference in the number of tables accessed by the view at compile-time and at run-time.

- In this example, the query expressions of the view differ because the projection lists at compile-time and run-time are different. The projection lists at compile-time and run-time contains I, J, A and I+1, J+2, B, respectively.

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT I, J, A
FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT I+1, J+2, B
FROM PRDCAT.PRDSCH_ONE.TABLE_ONE T1,
PRDCAT.PRDSCH_ONE.TABLE_TWO T2;
```

Thus, similarity check for the view fails.

- In this example, the query expressions of the view differs because of the difference in projection lists at compile-time and run-time. The projection lists at compile-time and run-time contains I, J, A and I, J, CURRENT_DATE AS A, respectively.

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT I, J, A FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO
T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT I, J, CURRENT_DATE AS A
FROM PRDCAT.PRDSCH_ONE.TABLE_ONE T1,
PRDCAT.PRDSCH_ONE.TABLE_TWO T2;
```

Thus, similarity check for the view fails.

- In this example, except for the table names, the query expression remains the same between compile-time and run-time. Therefore, the view definition remains unchanged, and the view similarity check is passed.

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT T1.X1, T1.Y1, X2
FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
SELECT T1.X1, T2.Y1, X2
FROM PRDCAT.PRDSCH_ONE.TABLE_T1 T1,
PRDCAT.PRDSCH_ONE.TABLE_T2 T2;
```

- In this example, the view similarity check fails because T1 and T2 are correlations pointing to different tables at compile-time and run-time.

Compile-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK
AS SELECT I, J, A
FROM CAT.SCH_ONE.TABLE_ONE T1, CAT.SCH_ONE.TABLE_TWO T2;
```

Run-time view:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK
AS SELECT I, J, A
FROM PRDCAT.PRDSCH.TABLE_ONE T2, PRDCAT.PRDSCH_ONE.TABLE_TWO
T1;
```

- In this example, the view similarity check fails because the correlation names for the tables, at compile-time, TAB1 and TAB2 and at run-time, TEMP1 and TEMP2 are not the same.

Compile-time view:

```
CREATE VIEW TEMP_VIEW
ENABLE SIMILARITY CHECK AS
SELECT A FROM
  (SELECT A FROM
    (SELECT A FROM CAT.SCH_ONE.TABLE_T4
     WHERE A > 1) TAB1,
    WHERE A < 10) TAB2
WHERE A <> 4;
```

Run-time view:

```
CREATE VIEW PRDTEMP _VIEW
ENABLE SIMILARITY CHECK AS
SELECT A FROM
  (SELECT A FROM
    (SELECT A FROM PRDCAT.PRDSCH_ONE.TABLE_T4
     WHERE A > 1) TEMP1,
    WHERE A < 10) TEMP2
WHERE A <> 4;
```

- In the following example, the correct table mapping for the view similarity check must be CAT.SCH.TABLE_ONE to PRDCAT.PRDSCH_ONE.TAB1 and CAT.SCH.TABLE_ONE to PRDCAT.PRDSCH_ONE.TAB1.

Compile-time view:

```
CREATE VIEW TEMP_VIEW ENABLE SIMILARITY CHECK AS
  SELECT I, J, A, B
    FROM CAT.SCH.TABLE_ONE T1, CAT.SCH.TABLE_TWO T2
   WHERE T1.I = T2.A;
```

Run-time view 1:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
  SELECT I, J, A, B
    FROM PRDCAT.PRDSCH_ONE.TAB2 T1, PRDCAT.PRDSCH_ONE.TAB1 T2
   WHERE T2.I = T1.A;
```

Run-time view 2:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
  SELECT I, J, A, B
    FROM PRDCAT.PRDSCH_ONE.TAB2 T2, PRDCAT.PRDSCH_ONE.TAB1 T1
   WHERE T1.I = T2.A;
```

Run-time view 3:

```
CREATE VIEW PRDTEMP_VIEW ENABLE SIMILARITY CHECK AS
  SELECT I, J, A, B
    FROM PRDCAT.PRDSCH_ONE.TAB1 T1, PRDCAT.PRDSCH_ONE.TAB2 T2
   WHERE T1.I = T2.A;
```

Run-time view 1 fails the view similarity check because the view text is different compared to the compile-time view, which is caused by a change in the position of the predicate.

Run-time view 2 fails the view similarity check because the positions of the correlation names (aliases) of the tables in the run-time view are different compared to the compile-time view, which causes a difference in the view text.

In addition, run-time view 1 and view 2 definitions also result in wrong SQL/MX tables being compared for the view similarity check.

Although the run-time views, view 1 and view 2 are semantically equivalent to the compile-time view, they fail the view similarity check because of the difference in the view text.

In run-time view 3, although the table names do not match, the view text matches with the compile-time. Run-time view 3 also results in correct mapping of underlying tables for the view similarity check. Therefore, the view similarity check is passed.

- In this example, run-time view 1 passes the view similarity check, because PRDCAT.SCH_ONE.tab1 is mapped to CAT.SCH.TABLE_ONE, and PRDCAT.SCH_ONE.tab2 is mapped to CAT.SCH.TABLE_TWO correctly.

Compile-time view:

```
CREATE VIEW PRDTEMP_VIEW AS
  SELECT I, J, B
    FROM CAT.SCH.TABLE_ONE, CAT.SCH.TABLE_TWO
   WHERE TABLE_TWO.B = 'A' AND TABLE_ONE.B = 1;
```

Run-time view 1:

```
CREATE VIEW PRDTEMP_VIEW AS
  SELECT I, J, B
    FROM PRDCAT.PRDSCH_ONE.TAB1, PRDCAT.PRDSCH_ONE.TAB2
   WHERE TAB2.B = 'A' AND TAB1.I = 1;
```

Run-time view 2:

```
CREATE VIEW PRDTEMP_VIEW AS
  SELECT I, J, B
    FROM PRDCAT.PRDSCH_ONE.TAB1, PRDCAT.PRDSCH_ONE.TAB2
   WHERE TAB1.I = 1 AND TAB2.B = 'A';
```

However, in run-time view 2, positions of the predicates do not match the compile-time view. Since the positions do not match, a difference exists between the view text in addition to the difference in the table names. Therefore, run-time view 2 fails the view similarity check.

If a statement fails the similarity check, the SQL/MX executor, by default, invokes the SQL/MX compiler to automatically recompile the SQL plan. For more information, see [Automatic Recompilation](#) on page 8-18.

Automatic Recompilation

Automatic recompilation is the run-time recompilation, invoked by the SQL/MX executor, of a DML statement in a module. During automatic recompilation, the SQL plan changes but is not written to the module. Instead, it is stored in the memory of the SQL/MX executor.

Automatic recompilation incurs a performance cost because it requires the query execution plan to be regenerated at run time and stored in memory. Automatically recompiled query plans are not saved for subsequent executions of the same program or for multiple concurrent executions of the same program. Because of this limitation, automatic recompilation might be unsuitable for some production environments.

Causes of Automatic Recompilation

By default, automatic recompilation is enabled for all embedded SQL programs. Automatic recompilation occurs if:

- The value of a class MAP DEFINE or PROTOTYPE host variable changes or the timestamp of a table changes, and the similarity check fails or is disabled. For more information, see [Similarity Check](#) on page 8-9.
- A DML statement was not compiled when you explicitly SQL compiled the module definition because the table did not exist or was unavailable at that time. For more information, see [Running the SQL/MX Compiler](#) on page 15-36.
- A transaction mode changes because of a SET TRANSACTION statement. For more information, see [Setting Attributes for Transactions](#) on page 14-3.

Controlling Automatic Recompilation

By default, automatic recompilation is enabled for all embedded SQL programs. To disable automatic recompilation at run time and force the explicit recompilation of DML statements, use the CONTROL QUERY DEFAULT statement with the AUTOMATIC_RECOMPILATION option set to OFF:

```
CONTROL QUERY DEFAULT AUTOMATIC_RECOMPILATION 'OFF' ;
```

Automatic recompilation remains OFF until the end of the embedded SQL program or until the occurrence of a CONTROL QUERY DEFAULT statement with AUTOMATIC_RECOMPILATION set to ON.

Controlling Automatic Recompilation Messages

By default, to comply with the ANSI standard, the SQL/MX executor does not return a warning message to the program when a DML statement is automatically recompiled. NonStop SQL/MX always logs a warning event, SQL/MX message 505, to the Event Management Service (EMS) log when a statement is automatically recompiled. For more information, see the *EMS Manual* and *Operator Messages Manual*.

To return recompilation warning messages directly to the program, use a CONTROL QUERY DEFAULT statement with the RECOMPILATION_WARNINGS option set to ON:

```
CONTROL QUERY DEFAULT RECOMPILATION_WARNINGS 'ON' ;
```

If the similarity check fails and RECOMPILATION_WARNINGS is ON, the SQL/MX executor returns warning message 8579 to the program. If a DML statement is automatically recompiled and RECOMPILATION_WARNINGS is ON, the SQL/MX executor returns warning message 8576 to the program.

Recommended Recompilation Settings for OLTP Programs

For optimal performance of OLTP programs in a production environment, use these settings:

Default Attribute	Value
SIMILARITY_CHECK	ON
RECOMPILATION_WARNINGS	OFF
AUTOMATIC_RECOMPILATION	OFF

A similarity check is faster than automatic recompilation and can reduce the performance costs of automatic recompilation. Therefore, you should enable similarity checks for OLTP programs.

Because automatic recompilation incurs a performance cost, it is unsuitable for OLTP programs and should be disabled. If you allow automatic recompilations but control the database environment to prevent them from occurring, you should monitor the EMS log for warning events that indicate that an automatic recompilation has occurred. You can also turn on the RECOMPILATION_WARNINGS option to report recompilation warning messages directly to the program when they occur.

For more information, see [Similarity Check](#) on page 8-9 and [Automatic Recompilation](#) on page 8-18.

9 Dynamic SQL

Using the dynamic SQL statements of NonStop SQL/MX, programs can construct, compile, and execute an SQL statement at run time. With static SQL, you code the actual SQL statement in the source file and compile the statement during explicit SQL compilation. A static SQL program uses host variables to send and receive values.

A dynamic SQL program, however, uses a character host variable as a placeholder for an SQL statement, which is usually unknown during explicit compilation. To construct the dynamic SQL statement in the host variable, the program usually requires some input from a user at a workstation. After receiving the input, the program constructs, compiles, and executes the dynamic SQL statement at run time.

Dynamic SQL can be useful for:

- **New interfaces:** Suppose that you need to develop an interactive interface similar to MXCI but designed for specific users. A dynamic SQL program can prompt the user for input, so the user does not need 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 form syntax elements.
- **Client-server support:** Suppose that your program is a server that receives requests from other programs. A program is created to manipulate data in a database. The program formulates an SQL statement and sends it to your server. Your server constructs, compiles, and executes the dynamic SQL statement and sends the results back to the program.

The statements you use to construct, compile, and execute SQL statements during run time are referred to as dynamic SQL.

The HP NonStop Connectivity Service (MXCS) uses dynamic SQL. For more information, see the *SQL/MX Connectivity Service Manual*.

This section describes:

- [Statements for Dynamic SQL With Arguments](#) on page 9-2
- [Input Parameters and Output Variables](#) on page 9-2
- [Steps for Using Dynamic SQL With Argument Lists](#) on page 9-3
- [Using EXECUTE IMMEDIATE](#) on page 9-7
- [Setting Default Values Dynamically](#) on page 9-8

For information on using dynamic SQL with descriptor areas, see [Section 10, Dynamic SQL With Descriptor Areas](#).

For information on using dynamic SQL cursors, see [Section 11, Dynamic SQL Cursors](#).

Statements for Dynamic SQL With Arguments

Some of the dynamic SQL statements commonly used in programs are:

PREPARE	Prepares (compiles) a dynamic SQL statement for subsequent execution by an EXECUTE statement.
DEALLOCATE PREPARE	Deallocates a prepared statement and returns the system resources used by the statement and also allows reuse of the statement name.
EXECUTE	Executes a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	Prepares (compiles) and executes a dynamic SQL statement in one step.

These statements are described on subsequent pages in this section. For the complete syntax of each statement, see the *SQL/MX Reference Manual*.

Input Parameters and Output Variables

An input parameter is a symbol in a dynamic SQL statement that serves as a placeholder for a value substituted when the statement executes. Input parameters are specified as question marks (?).

An input parameter can appear in an SQL expression wherever a constant can appear. Using a parameter, you can prepare an SQL statement without the input values. Specify the data type of the parameter explicitly by using the CAST function so that NonStop SQL/MX correctly types the parameter. The input values are then provided when the statement executes.

NonStop SQL/MX returns data to a program through output variables. Output variables are user-specified areas in the program. Output variables typically contain columns returned from a SELECT operation.

If you are using the form of the EXECUTE statement that provides a list of arguments in the USING and INTO clauses, you must know the nature of the dynamic input parameters and any SELECT list columns. The number of arguments and the data types of arguments provided in the EXECUTE statement must match both the number and the data types of parameters in the prepared statement.

If you do not know the number and data types of arguments, use the form of the EXECUTE statement that uses descriptor areas. See [Section 10, Dynamic SQL With Descriptor Areas](#).

Floating-Point Variables

Depending on the setting for the CONTROL QUERY DEFAULT FLOATTYPE statement, input and output will either be in IEEE FLOAT format or Tandem FLOAT format. The default value is Tandem FLOAT format for dynamic SELECT statements

and dynamic parameters. See the *SQL/MX Reference Manual* for information on setting CONTROL QUERY DEFAULT values.

Steps for Using Dynamic SQL With Argument Lists

[Figure 9-1](#) shows the steps presented within the complete C program. These steps are executed in the sample program [Example A-3](#) on page A-5.

C

Figure 9-1. Using Dynamic SQL in a C Program

```

1  ...
   EXEC SQL BEGIN DECLARE SECTION;
   char hv_sql_statement[256];
   long in_value;
   ... hv_column1;
   ... hv_column2;
   ...
   EXEC SQL END DECLARE SECTION;

2  ... /* Construct the SQL statement and move to statement variable. */
   ...

3  EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
   ...

4  ... /* Prompt user for value of input parameter. */
   scanf("%hu",&in_value);

5  EXEC SQL EXECUTE sqlstmt USING :in_value
   INTO :hv_column1,:hv_column2,...;

6  EXEC SQL DEALLOCATE PREPARE sqlstmt;

```

[Figure 9-2](#) shows the steps presented within the complete COBOL program. These steps are executed in the sample program [Example C-3](#) on page C-6.

COBOL**Figure 9-2. Using Dynamic SQL in a COBOL Program**

```

1  ...
    EXEC SQL BEGIN DECLARE SECTION;
    01 HV-SQL-STATEMENT    PIC X(256).
    01 IN-VALUE            PIC 9(5) COMP.
    01 HV-COLUMN1          ...
    01 HV-COLUMN2          ...
    ...
    EXEC SQL END DECLARE SECTION;

2  * Construct the SQL statement and move to statement variable.
    ...

3      EXEC SQL PREPARE sqlstmt FROM :HV-SQL-STATEMENT END-EXEC.
    ...

4  * Prompt user for value of input parameter.
    ...
    ACCEPT IN-VALUE.

5      EXEC SQL EXECUTE sqlstmt USING :IN-VALUE
        INTO :HV-COLUMN1,:HV-COLUMN2,... END-EXEC.

6      EXEC SQL DEALLOCATE PREPARE sqlstmt END-EXEC.
    ...

```

For more information:

1. [Declare a Host Variable for the Dynamic SQL Statement](#) on page 9-4
2. [Move the Statement Into the Host Variable](#) on page 9-5
3. [Prepare the SQL Statement](#) on page 9-5
4. [Set Explicit Input Values](#) on page 9-6
5. [Execute the Prepared Statement](#) on page 9-6
6. [Deallocate the Prepared Statement](#) on page 9-7

Declare a Host Variable for the Dynamic SQL Statement

In an SQL Declare Section, declare a host variable to use as a container for the dynamic SQL statement. You specify this host variable when you prepare the SQL statement. You must declare the host variable:

- Before the PREPARE statement
- Within the same scope as the PREPARE statement

Move the Statement Into the Host Variable

Move a statement containing parameters into the statement host variable. For example, the host variable named `hv_sql_statement` might contain a statement of this form:

```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, SALARY
FROM SAMDBCAT.PERSNL.EMPLOYEE
WHERE EMPNUM = CAST(? AS NUMERIC(4) UNSIGNED)
```

The parameter in this `SELECT` statement represents a value to be provided by the user. Providing the value of the primary key within a loop in your program is a typical example of user input for a dynamic SQL statement after the statement has been prepared.

Use the `CAST` function for a dynamic input parameter to ensure the data type of the parameter is the same as the data type of the host variable declared to hold the input value for the parameter. For this example, because `NUMERIC(4) UNSIGNED` is the data type of the employee number in the `EMPLOYEE` table, specify this data type for the parameter in the `AS` clause of the `CAST` function.

Examples

```
C strcpy(hv_sql_stmt, "SELECT empnum, first_name,"
        " last_name, salary"
        " FROM samdbcat.persnl.employee"
        " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)");
```

```
COBOL MOVE "SELECT empnum, first_name, last_name, salary"
        & " FROM samdbcat.persnl.employee"
        & " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)"
        TO hv-sql-stmt.
```

Prepare the SQL Statement

To execute the dynamic SQL statement, you must first prepare the statement that is stored in a host variable. The `PREPARE` statement checks the statement syntax, determines the data types of any parameters, and compiles the statement. The `PREPARE` statement also associates the prepared statement with a name that you can use in subsequent `EXECUTE` statements.

Use this general syntax:

```
PREPARE SQL-statement-name FROM :SQL-statement-variable
```

For complete syntax, see the `PREPARE` statement in the *SQL/MX Reference Manual*.

Example

```
C EXEC SQL PREPARE sqlstmt FROM: hv_sql_statement;
```

The SQL identifier `sqlstmt` is the name of the prepared statement to be used in a subsequent EXECUTE statement. The host variable `hv_sql_statement` contains the dynamic SQL statement.

Set Explicit Input Values

If you have dynamic input parameters in your prepared statement, you must code the appropriate C statements to prompt the user to input values and then store these values in the appropriate host variables.

Examples

C

```
/* Initialize input variable in WHERE clause. */
printf("Enter the employee number:");
scanf("%hu",&in_empnum);
```

COBOL

```
* Initialize input variable in WHERE clause.
  DISPLAY "Enter the employee number: ".
  ACCEPT IN-EMPNUM.
```

Execute the Prepared Statement

You are ready to execute the prepared SQL statement. The EXECUTE statement names the input parameters and output variables.

Use this general syntax:

```
EXECUTE SQL-statement-name
  USING variable-spec [,variable-spec]...
  INTO variable-spec [,variable-spec]...;
```

Before performing this operation, the application must store information for each input parameter of the prepared statement in the appropriate host variable. If you have more than one input parameter, supply the host variables for the parameters in the USING list in the order of the parameters' position in the prepared SQL statement.

When EXECUTE with INTO executes, NonStop SQL/MX stores information into the host variables (and optionally their indicator variables) that correspond to columns specified in the select list for the prepared statement. If you have more than one output variable, supply the host variables for the output variables in the INTO list in the order of the variables' position in the prepared SQL statement.

The number of arguments and the data types of arguments you provide in the EXECUTE statement must match the number of parameters and the data types of parameters in the prepared statement.

For complete syntax, see the EXECUTE statement in the *SQL/MX Reference Manual*.

Example

This statement uses both input parameters and output variables:

```
C EXEC SQL EXECUTE sqlstmt
      USING :in_empnum
      INTO  :hv_empnum, :hv_firstname, :hv_lastname,
            :hv_salary INDICATOR :hv_salary_i;
```

You must specify the indicator variable in the INTO argument list for columns that allow null.

Deallocate the Prepared Statement

When you are finished with the dynamic SQL statement, deallocate the resources used by the prepared statement. See the DEALLOCATE PREPARE statement in the *SQL/MX Reference Manual*.

Example

```
C EXEC SQL DEALLOCATE PREPARE sqlstmt;
... 
```

Using EXECUTE IMMEDIATE

If the dynamic SQL statement does not contain input or output parameters and you are planning to execute it only once, use the EXECUTE IMMEDIATE statement to prepare and execute in one step. In this case, the user provides the SQL statement. Otherwise, you could code the SQL statement as static SQL in your program.

Use this general syntax:

EXECUTE IMMEDIATE <i>SQL-statement-variable</i>

For complete syntax, see the EXECUTE IMMEDIATE statement in the *SQL/MX Reference Manual*.

You must specify the host variable that contains the statement to be prepared and executed in the EXECUTE IMMEDIATE statement.

Example

```
C strcpy (hv_sql_statement, "UPDATE employee"
      " SET salary = salary * 1.1"
      " WHERE jobcode = 1234");
```

```
EXEC SQL EXECUTE IMMEDIATE :hv_sql_statement;
```

The host variable `hv_sql_statement` contains the SQL statement.

Setting Default Values Dynamically

Use the EXECUTE IMMEDIATE statement to set dynamic SQL default values; for example, to set the WARN attribute to the default value `iud_nonaudited_index_maint`.

Examples

```
C strcpy(hv_sql_statement, "control query default  
iud_nonaudited_index_maint 'warn'");  
  
EXEC SQL EXECUTE IMMEDIATE :hv_sql_statement;
```

```
COBOL move "control query default iud_nonaudited_index_maint  
'warn'" to hv-sql-statement.  
  
EXEC SQL EXECUTE IMMEDIATE :hv-sql-statement END-EXEC.
```


Dynamic SQL With Descriptor Areas

Use dynamic SQL statements to construct, compile, and execute SQL statements during run time. Use the descriptor areas of NonStop SQL/MX to store information on each input parameter and output variable in a dynamic statement.

This section describes:

- [Statements for Dynamic SQL With Descriptors](#) on page 10-1
- [SQL Descriptor Areas](#) on page 10-2
- [Input Parameters](#) on page 10-3
- [Output Variables](#) on page 10-7
- [Steps for Using SQL Item Descriptor Areas](#) on page 10-12
- [Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data](#) on page 10-21
- [Using SQL Descriptor Areas to Retrieve ISO88591 Data to UCS2 Host Variables](#) on page 10-21

Statements for Dynamic SQL With Descriptors

These statements for dynamic SQL use descriptor areas:

ALLOCATE DESCRIPTOR	Allocates an input or output SQL descriptor area.
DEALLOCATE DESCRIPTOR	Deallocates an SQL descriptor area.
GET DESCRIPTOR	Retrieves information from an SQL descriptor area—the COUNT of the item descriptor areas and specified fields in the areas.
SET DESCRIPTOR	Modifies information in an SQL descriptor area.
PREPARE	Prepares (compiles) a dynamic SQL statement for subsequent execution by an EXECUTE statement.
DEALLOCATE PREPARE	Deallocates a prepared statement and returns the system resources used by the statement. It also allows you to reuse the name of the statement.
DESCRIBE INPUT	Stores in the descriptor area information on dynamic input parameters for a prepared statement.
DESCRIBE [OUTPUT]	Stores in the descriptor area information on output values (usually SELECT columns) from a prepared statement.
EXECUTE	Executes a prepared dynamic SQL statement.
EXECUTE IMMEDIATE	Prepares (compiles) and executes a dynamic SQL statement.

These statements are described on subsequent pages in this section. For the complete syntax of each statement, see the *SQL/MX Reference Manual*.

SQL Descriptor Areas

An SQL descriptor area consists of multiple item descriptor areas, together with a COUNT of the number of those item descriptor areas. You can use an input descriptor area to store information on input parameters and an output descriptor area to store information on output variables in your dynamic SQL statement.

When using SQL descriptor areas, note:

- To identify an SQL descriptor area, use an SQL identifier.
- To allocate an SQL descriptor area, use the ALLOCATE DESCRIPTOR statement.
- To provide the necessary information in the input SQL descriptor area to applications, use the DESCRIBE INPUT statement to describe the input parameters of a dynamic SQL statement.
- After you describe the parameters, use the SET DESCRIPTOR statement to set the input values for the parameters in the input SQL descriptor area. Alternately, you can use SET DESCRIPTOR to describe the input parameters explicitly (without using DESCRIBE INPUT) and set the input values.
- Output variables typically contain columns returned from a SELECT operation. Use the DESCRIBE OUTPUT statement to describe information in the output SQL descriptor area about the output variables.
- After you describe the output variables, use the GET DESCRIPTOR statement to retrieve information on them from the output SQL descriptor area.
- To deallocate an SQL descriptor area, use the DEALLOCATE DESCRIPTOR statement.

SQL Item Descriptors

NonStop SQL/MX uses an input descriptor area to store information on input parameters and an output descriptor area to store information on output variables in dynamic SQL statements. Each descriptor area consists a number of item descriptors. A sufficient number of item descriptors is required to store information on all the parameters and variables in your dynamic SQL statement.

To store information on input parameters in the descriptor area, use the DESCRIBE INPUT statement and the SET DESCRIPTOR statement.

To store information on output variables in the descriptor area, use the DESCRIBE OUTPUT statement. To retrieve information on output variables from the descriptor area, use the GET DESCRIPTOR statement.

Allocating an SQL Descriptor Area

Use the `ALLOCATE DESCRIPTOR` statement to allocate a named SQL descriptor area to store information necessary for the execution of dynamic SQL statements. If needed, allocate two descriptor areas: one for input parameters and one for output variables.

Use this general syntax:

```
ALLOCATE DESCRIPTOR descriptor-name WITH MAX occurrences
```

For complete syntax, see the `ALLOCATE DESCRIPTOR` statement in the *SQL/MX Reference Manual*.

The descriptor name is a literal or a host variable with a character data type that specifies an SQL identifier at run time. The number of occurrences is specified to be a host variable whose value is the maximum number of item descriptors, or the number of parameters to be used in the dynamic SQL statements in your program.

Deallocating an SQL Descriptor Area

After the execution of dynamic SQL statements in your program, use the `DEALLOCATE DESCRIPTOR` statement to deallocate an SQL descriptor area previously allocated with `ALLOCATE DESCRIPTOR`.

Use this general syntax:

```
DEALLOCATE DESCRIPTOR descriptor-name
```

For complete syntax, see the `DEALLOCATE DESCRIPTOR` statement in the *SQL/MX Reference Manual*.

The descriptor name for both the `ALLOCATE DESCRIPTOR` and `DEALLOCATE DESCRIPTOR` statements includes specifying a `GLOBAL` or `LOCAL` scope. The scope must be the same for these two statements using the same descriptor name within the same module or compilation unit.

Input Parameters

An input parameter is a symbol in a dynamic SQL statement that serves as a placeholder for a value substituted when the statement executes.

Specify input parameters in the statement as question marks (`?`), and use them in SQL expressions wherever a constant is valid. Using a parameter, you can prepare an SQL statement without the input values, which are substituted when the statement executes. Specify the data type of the parameter explicitly by using the `CAST` function so that NonStop SQL/MX correctly types the parameter.

Use the DESCRIBE INPUT statement to set information in the descriptor area about the input parameters. Alternatively, you can use the SET DESCRIPTOR statement to set information explicitly in the descriptor area for individual input parameters.

Describing Input Parameters

All values in the item descriptor areas are initially undefined. You can use a DESCRIBE INPUT statement to set information on the input parameters in the descriptor area.

Use this general syntax:

```
DESCRIBE INPUT statement-name
      USING SQL DESCRIPTOR descriptor-name
```

For complete syntax, see the DESCRIBE statement in the *SQL/MX Reference Manual*.

When DESCRIBE INPUT executes, NonStop SQL/MX stores information for each input parameter of the prepared statement in an item descriptor. Each parameter has a separate area.

The DESCRIBE INPUT statement sets all fields in the item descriptor for each input parameter, except for the VARIABLE_POINTER, VARIABLE_DATA, INDICATOR_POINTER, and INDICATOR_DATA fields. COUNT is set equal to the number of input parameters. For limitations on using the VARIABLE_POINTER item, see the VARIABLE POINTER in the *SQL/MX Reference Manual*.

Setting the Data Values of Input Parameters

After you describe input parameter values for a prepared statement, use the SET DESCRIPTOR statement to set the value of an input parameter in an item descriptor.

Use this general syntax:

```
SET DESCRIPTOR descriptor-name
  { COUNT = value-specification }
  { VALUE item-number desc-item-name = value-spec, ... }
```

For complete syntax, see the SET DESCRIPTOR statement in the *SQL/MX Reference Manual*.

The named descriptor area must be currently allocated. You can assign the input value to the VARIABLE_DATA or VARIABLE_POINTER field in the SQL descriptor area allocated earlier and then use SET DESCRIPTOR. For limitations on using the VARIABLE_POINTER item, see the SET DESCRIPTOR statement in the *SQL/MX Reference Manual*.

This example uses a typical context for input parameters in dynamic SQL. Specify the data type of the input parameter by using the CAST function so that DESCRIBE INPUT correctly types the parameter.

Example**C**

```

strcpy (hv_sql_statement, "UPDATE employee"
        " SET salary = salary * 1.1"
        " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)");

...
desc_max=1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
...
EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
...
EXEC SQL DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR 'in_sqlda';
...
scanf ("%hu",&in_jobcode);
...
desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
        VARIABLE_DATA = :in_jobcode;
...
EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda';

```

COBOL

```

MOVE "UPDATE employee"
-   " SET salary = salary * 1.1"
-   " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)"
  TO hv-sql-statement.

...
MOVE 1 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda'
        WITH MAX :desc-max
END-EXEC.

...
EXEC SQL PREPARE sqlstmt FROM :hv-sql-statement END-EXEC.
...
EXEC SQL DESCRIBE INPUT sqlstmt
        USING SQL DESCRIPTOR 'in_sqlda'
END-EXEC.

...
ACCEPT in-jobcode.

...
MOVE 1 TO desc-value.
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc-value
        VARIABLE_DATA = :in-jobcode
END-EXEC.

...
EXEC SQL EXECUTE sqlstmt
        USING SQL DESCRIPTOR 'in_sqlda'
END-EXEC.

```

Setting Input Parameter Information Without DESCRIBE INPUT

Use the SET DESCRIPTOR statement to describe the input parameters explicitly, without using DESCRIBE INPUT. To use this method, set the values for these fields (if needed for the particular data type):

```
TYPE_FS (or TYPE)
DATETIME_CODE
LEADING_PRECISION
PRECISION
SCALE
CHARACTER_SET_NAME
LENGTH
INDICATOR_TYPE
INDICATOR_DATA (or INDICATOR_POINTER)
VARIABLE_DATA (or VARIABLE_POINTER)
ROWSET_VAR_LAYOUT_SIZE
ROWSET_IND_LAYOUT_SIZE
```

The data type of the input parameter determines the fields that need to be set.

Example

This example uses a typical context for input parameters in dynamic SQL. This example does not use the DESCRIBE INPUT statement.

C

```
strcpy(hv_sql_stmt, "UPDATE samdbcat.persnl.employee"
      " SET salary = salary * 1.1"
      " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)");

/* Allocate the descriptor area for input parameters. */
desc_max=1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
...
/* Prepare the statement. */
EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;
...
scanf("%hu", &in_jobcode);
...
/* Set the value of the input parameters in */
/* the input SQL descriptor area. */
desc_value      = 1;
desc_type       = -502;          /* Smallint unsigned. */
desc_precision  = 4;
desc_scale      = 0;

EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
      TYPE          = :desc_type,
      PRECISION     = :desc_precision,
      SCALE         = :desc_scale,
      VARIABLE_DATA = :in_jobcode;
...
EXEC SQL EXECUTE sqlstmt
      USING SQL DESCRIPTOR 'in_sqlda';
```

For multiple input parameters, set the individual descriptors in the order of their occurrence in the dynamic SQL statement. For a complete example of this method, see [Example A-5](#) on page A-12.

Output Variables

NonStop SQL/MX returns data to a program through output variables. Output variables can be host variables or individual data buffers to which the program (through the SQL descriptor area) contains pointers. Output variables typically contain columns returned from a SELECT or FETCH operation. A program uses the DESCRIBE statement to set information on the output variables in the output descriptor area.

Describing Output Variables

Use a DESCRIBE OUTPUT statement to set output values for a prepared SELECT statement. Use GET DESCRIPTOR to retrieve the values.

Use this general syntax:

```
DESCRIBE [OUTPUT] statement-name
        USING SQL DESCRIPTOR descriptor-name
```

For complete syntax, see the DESCRIBE statement in the *SQL/MX Reference Manual*.

When DESCRIBE OUTPUT executes, NonStop SQL/MX stores the value of each output variable of the prepared statement in an item descriptor area. Each output value has a separate descriptor area.

Getting the Values of Output Variables

Use the GET DESCRIPTOR statement to retrieve the number (count) of item descriptor areas and to retrieve the value of an output variable in a specific item descriptor area.

Use this general syntax:

```
GET DESCRIPTOR descriptor-name
{ variable-name = COUNT }
{ VALUE item-number variable-name = desc-item-name, ... }
```

For complete syntax, see the GET DESCRIPTOR statement in the *SQL/MX Reference Manual*.

The DESCRIBE OUTPUT statement sets fields in the SQL item descriptor area for every column in the select list of a dynamic SELECT statement or in the select list of a cursor specification for a dynamic SQL cursor. COUNT is set equal to the number of columns. The NAME field contains the name of each column. The TYPE field contains the data type of each column.

If you are retrieving the value of an output variable from the VARIABLE_DATA field in the descriptor area, the receiving host variable must be of a compatible data type and size for the information being retrieved. Retrieve the output values in the VARIABLE_DATA field by using the GET_DESCRIPTOR statement within the COUNT loop and by testing on TYPE or NAME to assign the value to the host variable. In addition, you cannot use arithmetical computation with the VARIABLE_DATA field.

Typically, columns that are not known to the static compiled program are fetched by a dynamic SQL cursor when the cursor specification is provided by the user. For an example of this method, see [Section 11, Dynamic SQL Cursors](#).

Consideration—Retrieving Multiple Values From a Large Buffer

To retrieve multiple values efficiently from a large buffer, consider using VARIABLE_POINTER. In this case, use GET_DESCRIPTOR once to retrieve pointers to individual values. Otherwise, you must use GET_DESCRIPTOR multiple times to retrieve each individual value.

A typical statement sequence using VARIABLE_POINTER consists of:

1. DESCRIBE_OUTPUT statement
2. SET_DESCRIPTOR statement with VARIABLE_POINTER (set up pointers to individual values)
3. EXECUTE statement
4. FETCH statement (obtain data)
5. GET_DESCRIPTOR statement (obtain pointers to data)

VARIABLE_POINTER is described in the SET_DESCRIPTOR statement in the *SQL/MX Reference Manual*. [Example 10-1](#) describes how to use VARIABLE_POINTER in a C program.

C**Example 10-1. C VARIABLE_POINTER Example** (page 1 of 3)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void assign_to_hv(void);
void sql_error(void);
int main(void)
{
    char SQLSTATE_NODATA[6] = "02000";
    EXEC SQL BEGIN DECLARE SECTION;
        char SQLSTATE[6];
        char hv_sql_statement[1024];
        char hv_sqlstmt[30];
        char hv_curspec[256];
        int desc_max;
        long hv_num;
        VARCHAR hv_sqlda_name[129];
        long hv_type;
        long hv_empnum_length;
        long hv_empname_length;
        long hv_empnum_ptr;
        long hv_empname_ptr;
        unsigned short hv_i;
    EXEC SQL END DECLARE SECTION;
    SQLSTATE[5] = '\0';
    SQLSTATE_NODATA[5] = '\0';
    EXEC SQL WHENEVER SQLERROR CALL sql_error

;

    EXEC SQL DECLARE CATALOG 'seg';
    EXEC SQL DECLARE SCHEMA 'suzuki';
    strcpy(hv_sqlstmt, "hvstmt");
    strcpy(hv_curspec, "hvcur");
    strcpy(hv_sql_statement,
        "SELECT EMP_NO, EMP_NAME FROM EMP
        WHERE EMP_NO = '177397 '");
    EXEC SQL PREPARE :hv_sqlstmt FROM :hv_sql_statement;
    desc_max = 10;
    EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda'
        WITH MAX :desc_max;

```

C**Example 10-1. C VARIABLE_POINTER Example (page 2 of 3)**

```

EXEC SQL DESCRIBE OUTPUT :hv_sqlstmt
      USING SQL DESCRIPTOR 'out_sqlda';
EXEC SQL DECLARE :hv_curspec CURSOR FOR :hv_sqlstmt;
EXEC SQL OPEN :hv_curspec
      USING SQL DESCRIPTOR 'in_sqlda';
EXEC SQL GET DESCRIPTOR 'out_sqlda' :hv_num = COUNT;
printf ("Number of columns = %d\n", hv_num);
// Get the type and length of 1st column
hv_i =1;
EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE:hv_i
      :hv_empnum_length = LENGTH,
      :hv_type = TYPE;

char *hv_emp_num;
hv_emp_num = new char [hv_empnum_length];
hv_empnum_ptr = (long)hv_emp_num;
EXEC SQL SET DESCRIPTOR 'out_sqlda' VALUE :hv_i
VARIABLE_POINTER =:hv_empnum_ptr;
// Get the type and length of the second column
hv_i = 2;
EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :hv_i
      :hv_empname_length = LENGTH,
      :hv_type = TYPE;

char *hv_emp_name;
hv_emp_name = new char [hv_empname_length];
hv_empname_ptr = (long)hv_emp_name;
/* Just setting the pointers to retrieve column info */
EXEC SQL SET DESCRIPTOR 'out_sqlda' VALUE :hv_i
VARIABLE_POINTER =:hv_empname_ptr;
EXEC SQL FETCH :hv_curspec
      INTO SQL DESCRIPTOR 'out_sqlda';
while ((strcmp(SQLSTATE,SQLSTATE_NODATA) != 0)
      &&
      (strcmp(SQLSTATE,"24000") != 0)
      )
{
    printf("[Column-1] Length = %d ",
      hv_empnum_length);
    hv_emp_num[hv_empnum_length] = '\0';
    printf(" VARIABLE POINTER CONTAINS  = %s\n",
      hv_emp_num);
    printf("[Column-2] Length  =%d  ",
      hv_empname_length);
    hv_emp_name[hv_empname_length] = '\0';
    printf(" VARIABLE POINTER CONTAINS  = %s\n",
      hv_emp_name);
    EXEC SQL FETCH :hv_curspec
      INTO SQL DESCRIPTOR 'out_sqlda';
} /* while */

```

C**Example 10-1. C VARIABLE_POINTER Example** (page 3 of 3)

```

        delete hv_emp_num;
delete hv_emp_name;
EXEC SQL CLOSE :hv_curspec;
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';
EXEC SQL DEALLOCATE PREPARE :hv_sqlstmt;

return 0;
} /* main() */
void sql_error()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    long hv_num;
    unsigned short hv_i;
    char hv_sqlstate[6];
    VARCHAR hv_tabname[129];
    VARCHAR hv_colname[129];
    VARCHAR hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;
    printf("=== EXEC SQL WHENEVER SQLERROR CONTINUE ===\n");
    EXEC SQL GET DIAGNOSTICS :hv_num = NUMBER;

    for (hv_i=1; hv_i <= hv_num; hv_i++){
        EXEC SQL GET DIAGNOSTICS EXCEPTION :hv_i
            :hv_tabname = TABLE_NAME,
            :hv_colname = COLUMN_NAME,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_msgtxt = MESSAGE_TEXT;
        hv_tabname[128] = '\0';
        hv_colname[128] = '\0';
        hv_sqlstate[5] = '\0';
        hv_msgtxt[128] = '\0';
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE  : %s\n", hv_sqlstate);
        printf("message   : %s\n", hv_msgtxt);
    } /* for */
} /* sql_error() */

```

Steps for Using SQL Item Descriptor Areas

Figure 10-1 shows the steps presented within the complete C program. These steps are executed in the sample program [Example A-4](#) on page A-8.

C

Figure 10-1. Using SQL Descriptor Areas in a C Program

```

1 EXEC SQL BEGIN DECLARE SECTION;
   char hv_sql_statement[256];
   long in_value;
   short num;
   /* Declare host variables for item descriptor values.      */
   long num_data;
   char char_data[100];
   ...
EXEC SQL END DECLARE SECTION;

2 ...      /* Construct the SQL statement from user input.      */

3 desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
desc_max = 6;
EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :desc_max;

4 EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
   ...

5 EXEC SQL DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR 'in_sqlda';
EXEC SQL DESCRIBE OUTPUT sqlstmt USING SQL DESCRIPTOR 'out_sqlda';

6 desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
   VARIABLE_DATA = :in_value;

7 EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda'
   INTO SQL DESCRIPTOR 'out_sqlda';

8 EXEC SQL GET DESCRIPTOR 'out_sqlda' :num = COUNT;
for (i = 1; i<= num; i++) {
   EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
   :type = TYPE,
   :length = LENGTH,
   :name = NAME;
   /* Test type or name to determine the compatible host variable. */
   if ...
      EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
      :num_data = VARIABLE_DATA;
   ...
}
... /* Process the values from the item descriptor areas.      */

9 EXEC SQL DEALLOCATE PREPARE sqlstmt;
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda';
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';

```

[Figure 10-2](#) shows the steps presented within the complete COBOL program. These steps are executed in the sample program [Example C-4](#) on page C-9.

COBOL**Figure 10-2. Using SQL Descriptor Areas in a COBOL Program**

```

1      EXEC SQL BEGIN DECLARE SECTION;
      01 hv-sql-statement    PIC X(256).
      01 in-value           PIC 9(5) COMP.
      01 num                PIC 9(4) COMP.
      * Declare host variables for item descriptor values.
      01 num-data           PIC S9(9) COMP.
      01 char-data         PIC X(100).
      ...
      EXEC SQL END DECLARE SECTION;

2      * Construct the SQL statement from user input.
      ...

3      MOVE 1 TO desc-max.
      EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc-max ...
      MOVE 6 TO desc-max.
      EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :desc-max ...

4      EXEC SQL PREPARE sqlstmt FROM :hv-sql-statement END-EXEC.
      ...

5      EXEC SQL DESCRIBE INPUT sqlstmt
          USING SQL DESCRIPTOR 'in_sqlda' END-EXEC.
      EXEC SQL DESCRIBE OUTPUT sqlstmt
          USING SQL DESCRIPTOR 'out_sqlda' END-EXEC.

6      MOVE 1 TO desc-value.
      EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc-value
          VARIABLE_DATA = :in-value END-EXEC.

7      EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda'
          INTO SQL DESCRIPTOR 'out_sqlda' END-EXEC.

8      EXEC SQL GET DESCRIPTOR 'out_sqlda' :num = COUNT END-EXEC.
      PERFORM VARYING i FROM 1 BY 1 UNTIL i > num
          EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
              :type = TYPE,
              :length = LENGTH,
              :name = NAME
          END-EXEC.
      * Test type or name to determine the compatible host variable.
      if ...
          EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
              :num-data = VARIABLE_DATA
          END-EXEC.
      END-PERFORM.
      * Process the values from the item descriptor areas.

9      EXEC SQL DEALLOCATE PREPARE sqlstmt END-EXEC.
      EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda' END-EXEC.
      EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda' END-EXEC.

```

For more information:

1. [Declare a Host Variable for the Dynamic SQL Statement](#) on page 10-14
2. [Construct the SQL Statement From User Input](#) on page 10-14
3. [Allocate Input and Output SQL Descriptor Areas](#) on page 10-14

4. [Prepare the SQL Statement](#) on page 10-15
5. [Describe the Input Parameters and the Output Variables](#) on page 10-15
6. [Set Explicit Input Values](#) on page 10-16
7. [Execute the Prepared Statement](#) on page 10-18
8. [Get the Count and Descriptions of the Output Variables](#) on page 10-19
9. [Deallocate the Prepared Statement and the SQL Descriptor Areas](#) on page 10-20

Declare a Host Variable for the Dynamic SQL Statement

In an SQL Declare Section, declare a host variable to use as a container for the dynamic SQL statement. Specify this host variable when you prepare the SQL statement. You must declare the host variable:

- Before the PREPARE statement
- Within the same scope as the PREPARE statement

Construct the SQL Statement From User Input

Construct the SQL statement from user input and store the statement in the host variable. For example, the host variable named `hv_sql_statement` might contain a statement of this form:

```
SELECT column-list
FROM table
WHERE column = CAST(? AS S)
```

The variables (specified in italics) in the preceding SELECT statement represent character strings to be provided by the user.

Example

After specifying the table name, the column name, and the data type to be used for row selection, the host variable `hv_sql_statement` might contain this statement:

```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, SALARY
FROM SAMDBCAT.PERSNL.EMPLOYEE
WHERE EMPNUM = CAST(? AS NUMERIC(4) UNSIGNED)
```

Allocate Input and Output SQL Descriptor Areas

You must allocate an input SQL descriptor area for dynamic input parameters and an output SQL descriptor area for values of output variables (typically, columns of a SELECT statement). See [Allocating an SQL Descriptor Area](#) on page 10-3.

Allocating the Input SQL Descriptor Area

The number of item descriptor areas for the input SQL descriptor area must be large enough to accommodate all the dynamic input parameters of the prepared statement.

Example

C

```
desc_max = 1;  
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
```

This descriptor area can hold only one dynamic input parameter in the prepared statement.

Allocating the Output SQL Descriptor Area

The number of item descriptor areas for the output SQL descriptor area must be large enough to hold all the columns of the table referred to in the prepared statement.

Example

C

```
desc_max = 6;  
EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :desc_max;
```

This descriptor area can hold up to six columns selected from the table referred to in the prepared statement.

Prepare the SQL Statement

To execute the dynamic SQL statement, first prepare the statement you have constructed from user input and stored in a host variable. The PREPARE statement checks the statement syntax, determines the data types of any parameters, and compiles the statement. The PREPARE statement also associates the prepared statement with a name you can use in subsequent EXECUTE statements.

Use this general syntax:

```
PREPARE SQL-statement-name FROM :SQL-statement-variable
```

For complete syntax, see the PREPARE statement in the *SQL/MX Reference Manual*.

C

Example

```
EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
```

The SQL identifier *sqlstmt* is the name of the prepared statement to be used in a subsequent EXECUTE statement. The host variable *hv_sql_statement* contains the dynamic SQL statement you have constructed from user input.

Describe the Input Parameters and the Output Variables

To provide information necessary for the execution of the dynamic SQL statement, you must describe the dynamic input parameters and the output variables in the prepared statement. See [Describing Input Parameters](#) on page 10-4 and [Describing Output Variables](#) on page 10-7.

Describing the Input Parameters

Use the DESCRIBE INPUT statement to provide information for each dynamic input parameter in an item descriptor area, except for the actual VARIABLE_DATA and INDICATOR_DATA values.

Example

```
C EXEC SQL DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR 'in_sqlda';
```

Describing the Output Variables

Use the DESCRIBE OUTPUT statement to provide information for each possible output variable in an item descriptor area—typically, each column in the table referred to in a SELECT statement.

Example

```
C EXEC SQL DESCRIBE OUTPUT sqlstmt  
      USING SQL DESCRIPTOR 'out_sqlda';
```

If the table referred to consists of six columns, information is provided in six item descriptor areas. The COUNT for the SQL descriptor area is set to 6.

Set Explicit Input Values

If you have dynamic input parameters in your prepared statement, you must code the appropriate C statements to prompt the user to input the values and set the values in the descriptor area. See [Setting the Data Values of Input Parameters](#) on page 10-4.

Prompting the User

You first prompt the user for the input values.

Examples

```
C printf("Enter the employee number:");  
scanf("%hu",&in_empnum);
```

```
COBOL DISPLAY "Enter the employee number: ".  
ACCEPT IN-EMPNUM.
```


Setting the Parameter Values

After user input, you can set the VARIABLE_DATA or VARIABLE_POINTER value in the item descriptor area. For limitations on using the VARIABLE_POINTER item, see using the VARIABLE_POINTER in the *SQL/MX Reference Manual*.

Example

```
C desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
      VARIABLE_DATA = :in_empnum;
```

Setting Null

If the user can enter null, set the INDICATOR_DATA value to indicate null in the VARIABLE_DATA field in the item descriptor area.

Example

```
C desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
      INDICATOR_DATA = -1;
```

In this case, the actual value of VARIABLE_DATA is irrelevant and is not verified.

Setting Parameter Values by Position

If you have more than one dynamic parameter, you can identify and set values for the parameters by their position in the prepared SQL statement.

Example

```
C strcpy (hv_sql_statement, "UPDATE employee"
      " SET salary = salary * 1.1"
      " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)"
      " AND salary < CAST(? AS NUMERIC(8,2) UNSIGNED)");
...
desc_max = 2;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;
...
EXEC SQL PREPARE sqlstmt FROM :hv_sql_statement;
...
EXEC SQL DESCRIBE INPUT sqlstmt
      USING SQL DESCRIPTOR 'in_sqlda';
...
/* Input the values for the jobcode and salary parms. */
desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
      VARIABLE_DATA = :in_jobcode;
desc_value = 2;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
      VARIABLE_DATA = :in_salary;
```

```
...
EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda';
```

COBOL

```
...
MOVE "UPDATE employee"
-   " SET salary = salary * 1.1"
-   " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)"
-   " AND salary < CAST(? AS NUMERIC(8,2) UNSIGNED)"
  TO hv-sql-statement.

...
MOVE 2 TO desc-max.
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda'
      WITH MAX :desc_max
END-EXEC.

...
EXEC SQL PREPARE sqlstmt FROM :hv-sql-statement END-EXEC.
EXEC SQL DESCRIBE INPUT sqlstmt
      USING SQL DESCRIPTOR 'in_sqlda'
END-EXEC.

...
* Input the values for the jobcode and salary parms.
  ACCEPT in-jobcode.
  ACCEPT in-salary.

...
MOVE 1 TO desc-value.
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc-value
      VARIABLE_DATA = :in-jobcode
END-EXEC.
MOVE 2 TO desc-value.
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc-value
      VARIABLE_DATA = :in-salary
END-EXEC.

...
EXEC SQL EXECUTE sqlstmt
      USING SQL DESCRIPTOR 'in_sqlda'
END-EXEC.
```

Execute the Prepared Statement

You have allocated the input and output SQL descriptor area, described parameters, and set input values in SQL descriptor areas. You are ready to execute the prepared SQL statement. The EXECUTE statement names both the input SQL descriptor area and the output SQL descriptor area (if needed).

Use this general syntax:

```
EXECUTE SQL-statement-name
      USING SQL DESCRIPTOR in-descriptor-name
      INTO SQL DESCRIPTOR out-descriptor-name;
```

For complete syntax, see the EXECUTE statement in the *SQL/MX Reference Manual*.

Example

This statement uses both an input and output SQL descriptor area:

```
C EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR 'in_sqlda'
      INTO SQL DESCRIPTOR 'out_sqlda';
```

Get the Count and Descriptions of the Output Variables

When you have executed the prepared SQL statement, use the GET DESCRIPTOR statement to retrieve the output values from the output SQL descriptor area. See [Getting the Values of Output Variables](#) on page 10-7.

Retrieving the Count

Use the GET DESCRIPTOR statement to retrieve the COUNT of the SQL item descriptor areas.

Example

```
C EXEC SQL GET DESCRIPTOR 'out_sqlda' :num = COUNT;
```

Looping Through the Item Descriptor Areas

Use the COUNT to loop through the item descriptor areas. Test on TYPE or NAME to assign the value in VARIABLE_DATA to a compatible variable. You can use other descriptor fields, such as PRECISION, to modify this value. If a selected column is nullable, you must test the value in the INDICATOR_DATA field to determine whether an actual value for the column is retrieved in the VARIABLE_DATA field.

Example

```
C /* First, get the count of the number of output values. */
EXEC SQL GET DESCRIPTOR 'out_sqlda' :num = COUNT;
/* Second, get the i-th output values and save. */
for (i = 1; i <= num; i++) {
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
        :sqlda_type = TYPE,
        :sqlda_name = NAME;
    ...
    /* Test type or name to determine the host variable. */
    /* Assign data value to a compatible host variable. */
    ...
    if (strcmp(sqlda_name, "LAST_NAME", strlen("LAST_NAME"))==0){
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_last_name = VARIABLE_DATA;
        hv_last_name[20]='\0';
        printf("\nLast name is: %s", hv_last_name);
    }
    ...
    else
        if (strcmp(sqlda_name, "JOB CODE", strlen("JOB CODE"))==0){
            EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
```

```

        :hv_jobcode    = VARIABLE_DATA,
        :hv_jobcode_i = INDICATOR_DATA;
    if (hv_jobcode_i < 0)
        printf("\nJobcode is unknown");
    else
        printf("\nJobcode is: %hu", hv_jobcode);
    }
    ...
} /* end for */
...          /* process the item descriptor values */

```

COBOL

```

* First, get the count of the number of output values.
EXEC SQL
    GET DESCRIPTOR 'out_sqlda' :num = COUNT
END-EXEC.
* Second, get the i-th output values and save.
PERFORM VARYING i FROM 1 BY 1 UNTIL i > num
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
        :sqlda-type = TYPE,
        :sqlda-name = NAME
    END-EXEC.
* Test type or name to determine the host variable.
* Assign data value to a compatible host variable.
    ...
    IF sqlda-name = "LAST_NAME"
        EXEC SQL
            GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv-last-name = VARIABLE_DATA
        END-EXEC.
        DISPLAY "Last name is: " hv-last-name
    ...
    ELSE
        IF sqlda-name = "JOBCODE"
            EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
                :hv_jobcode    = VARIABLE_DATA,
                :hv_jobcode_i = INDICATOR_DATA
            END-EXEC.
            IF hv_jobcode_i < 0
                DISPLAY "Jobcode is unknown"
            ELSE
                DISPLAY "Jobcode is: " hv_jobcode
            ...
        END-PERFORM.
* Process the item descriptor values
    ...

```

Deallocate the Prepared Statement and the SQL Descriptor Areas

When you are finished with the dynamic SQL statement, deallocate the resources used by the prepared statement and the SQL descriptor areas. See the DEALLOCATE PREPARE statement in the *SQL/MX Reference Manual*.

Example**C**

```
EXEC SQL DEALLOCATE PREPARE sqlstmt;  
...  
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda';  
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';
```

Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data

See [Example A-9](#) on page A-28 and [Example A-10](#) on page A-30 for a detailed example.

Using SQL Descriptor Areas to Retrieve ISO88591 Data to UCS2 Host Variables

See [Example A-11](#) on page A-35 for a dynamic SQL descriptor example that retrieves ISO88591 data to UCS2 host variables.

11 Dynamic SQL Cursors

The dynamic SQL programs of NonStop SQL/MX use cursors to process multiple row SELECT statements in the same way that static SQL programs use cursors. The program reads rows from a result table, one by one, and sends the column values to output data buffers specified in the program. A static cursor is associated with an actual query expression (for example, a SELECT statement), and a dynamic cursor is associated with a statement prepared for the cursor specification.

This section describes:

- [Statements for Dynamic SQL Cursors](#) on page 11-1
- [Steps for Using a Dynamic SQL Cursor](#) on page 11-2
- [Dynamic SQL Cursors Using Descriptor Areas](#) on page 11-10

Statements for Dynamic SQL Cursors

PREPARE	Prepares (compiles) a dynamic SQL statement for subsequent execution.
DEALLOCATE PREPARE	Deallocates a prepared statement, returns the system resources used by the statement, and allows reuse of the statement name.
DECLARE CURSOR	Defines a cursor and associates it with a statement already prepared for the cursor specification.
ALLOCATE CURSOR	Allows you to dynamically declare an unlimited number of cursors.
OPEN	Opens a cursor.
FETCH	Positions a cursor on the next row of a table and retrieves values from the row.
Positioned UPDATE	Updates a row from a table or view at the current cursor position.
Positioned DELETE	Deletes a row from a table or view at the current cursor position.
CLOSE	Closes a cursor.

These statements are described on subsequent pages in this section. For the complete syntax of each statement, see the *SQL/MX Reference Manual*.

Steps for Using a Dynamic SQL Cursor

[Figure 11-1](#) shows the steps presented within the complete C program. These steps are executed in the sample program [Example A-6](#) on page A-15.

C

Figure 11-1. Using a Dynamic SQL Cursor in a C Program

```

1  ...
   EXEC SQL BEGIN DECLARE SECTION;
   short hostvar;
   char curspec[80];
   ...
   EXEC SQL END DECLARE SECTION;
   ...

2  strcpy(curspec, "SELECT column1, column2, column3"
               " FROM catalog.schema.table"
               " WHERE column1 = CAST(? AS sql_type)");
   ...
   EXEC SQL PREPARE cursor_spec FROM :curspec;
   ...

3  EXEC SQL DECLARE sql_cursor CURSOR FOR cursor_spec;
   ...
   void find_row(void)
   {
   ...

4  hostvar = initial_value;
   ...

5  EXEC SQL OPEN sql_cursor USING :hostvar;
   ...

6  EXEC SQL FETCH sql_cursor
   INTO :hostvar1, :hostvar2, :hostvar3 ;
   ...
7  ... /* Process values in the host variable(s). */
   ...
   ... /* If last row has not been processed, */
   ... /* branch back to fetch another row. */
8  ...

9  EXEC SQL CLOSE sql_cursor;
   ...
   }
   ...
   EXEC SQL DEALLOCATE PREPARE cursor_spec;
   ...

```


[Figure 11-2](#) shows the steps presented within the complete COBOL program. These steps are executed in the sample program [Example C-5](#) on page C-13.

COBOL**Figure 11-2. Using a Dynamic SQL Cursor in a COBOL Program**

```

1      ...
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 HOSTVAR      9(4) COMP.
01 CURSPEC      PIC X(80).
      ...
      EXEC SQL END DECLARE SECTION END-EXEC.
      ...

2      MOVE "SELECT column1, column2, column3"
-        " FROM catalog.schema.table"
-        " WHERE column1 = CAST(? AS sql_type)"
      TO CURSPEC.
      ...
      EXEC SQL PREPARE cursor_spec FROM :CURSPEC END-EXEC.
      ...

3      EXEC SQL DECLARE sql_cursor CURSOR FOR cursor_spec END-EXEC.
      ...

4      MOVE INITIAL-VALUE TO HOSTVAR.
      ...

5      EXEC SQL OPEN sql_cursor USING :HOSTVAR END-EXEC.
      ...

6      EXEC SQL FETCH sql_cursor
      INTO :HOSTVAR1, :HOSTVAR2, :HOSTVAR3 END-EXEC.
      ...
7      * Process values in the host variable(s).
      ...
      * If last row has not been processed,
      * branch back to fetch another row.
8      ...

9      EXEC SQL CLOSE sql_cursor END-EXEC.
      ...
      EXEC SQL DEALLOCATE PREPARE cursor_spec END-EXEC.
      ...

```

For more information:

1. [Declare Required Host Variables](#) on page 11-4
2. [Prepare the Cursor Specification](#) on page 11-4
3. [Declare the Cursor](#) on page 11-4
4. [Initialize the Dynamic Input Parameters](#) on page 11-5
5. [Open the Cursor](#) on page 11-5
6. [Retrieve the Values](#) on page 11-5
7. [Process the Retrieved Values](#) on page 11-6
8. [Fetch the Next Row](#) on page 11-6
9. [Close the Cursor and Deallocate the Prepared Statement](#) on page 11-6

Declare Required Host Variables

In an SQL Declare Section, declare the host variable you specify as the statement name for the cursor specification within the DECLARE CURSOR statement. You must also declare host variables used in the OPEN statement for dynamic input parameters. Declare required host variables:

- Before the PREPARE statement for the cursor specification
- Before the OPEN statement using the values for dynamic input parameters for the cursor specification
- Within the same scope as the SQL statements that refer to them

Prepare the Cursor Specification

In dynamic SQL, a host variable contains the query expression that serves as the cursor specification. Initialize the host variable before you execute the PREPARE statement for the cursor specification. The query expression might contain a dynamic input parameter.

After you initialize the host variable with the cursor specification, use the PREPARE statement to compile the cursor specification and associate the compiled statement with a statement name. Use the statement name in the DECLARE CURSOR statement.

Use this general syntax:

```
PREPARE statement-name FROM :cursor-specification
```

For complete syntax, see the PREPARE statement in the *SQL/MX Reference Manual*.

Declare the Cursor

A dynamic DECLARE CURSOR statement names a cursor and associates it with a statement name. Within a dynamic DECLARE CURSOR statement, both the cursor name and statement name can be provided at run time by the values of host variables. The statement name identifies the cursor specification already prepared within the same scope as the DECLARE CURSOR statement.

Use this general syntax:

```
DECLARE {cursor-name | ext-cursor-name}  
CURSOR FOR statement-name
```

For complete syntax, see the DECLARE CURSOR Declaration in the *SQL/MX Reference Manual*.

Code a dynamic DECLARE CURSOR statement in your program:

- In listing order, before other SQL statements that refer to the cursor, including the OPEN, FETCH, DELETE, UPDATE, and CLOSE statements
- Within the scope of other SQL statements that refer to the cursor

Initialize the Dynamic Input Parameters

Initialize the dynamic input parameters you specified in the cursor specification in the DECLARE CURSOR declaration.

You must initialize the host variables before you execute the OPEN statement. The OPEN statement uses the values of the input parameters to establish the result table and position the cursor before the first row of the table.

Open the Cursor

The OPEN statement determines the result table and positions the cursor before the first row of the table.

Use this general syntax:

```
OPEN cursor-name USING variable-spec [,variable-spec]...
```

For complete syntax, see the OPEN statement in the *SQL/MX Reference Manual*.

You can also use this syntax for an OPEN statement that uses an input descriptor area (that has been allocated, described, and initialized with the appropriate input parameter values):

```
OPEN cursor-name USING SQL DESCRIPTOR descriptor-name
```

Retrieve the Values

The FETCH statement positions the cursor at the next row of the result table and transfers the values defined in the query expression of the DECLARE CURSOR statement into the corresponding host variables.

Use this general syntax:

```
FETCH cursor-name INTO :hostvar [, :hostvar ]...
```

For complete syntax, see the FETCH statement in the *SQL/MX Reference Manual*.

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

Alternatively, use this general syntax when transferring values to an output descriptor area (that has been allocated and described):

```
FETCH cursor-name USING SQL DESCRIPTOR descriptor-name
```

For further information on using this method, see [Dynamic SQL Cursors Using Descriptor Areas](#) on page 11-10.

Process the Retrieved Values

After the FETCH statement returns the values to the host variables, your program can process the values. For example, you can test one or more values and then perform one of these operations:

- Update columns in the current row by using a positioned UPDATE statement.
- Delete the current row by using a positioned DELETE statement.
- List or display the values.
- Save the values in an array and process them later.

After you process a row, execute the FETCH statement to retrieve the next row.

Fetch the Next Row

Program control returns to the FETCH statement. Use the FETCH statement to position the cursor at the next row of the result table. Continue executing this loop until you have processed all rows specified by the query expression. After the FETCH statement has retrieved the last row, a subsequent FETCH causes a no-data exception (SQLSTATE is 02000 and SQLCODE is 100).

Close the Cursor and Deallocate the Prepared Statement

The CLOSE statement closes the cursor and releases the result table established by the OPEN statement. After the CLOSE statement executes, the result table no longer exists. To use the same cursor again, you must reopen it by using an OPEN statement. If the cursor specification contains a dynamic input parameter, the host variable in the USING clause of the OPEN statement can be initialized with a new value before the cursor is opened.

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

```
CLOSE cursor-name
```

For complete syntax, see the CLOSE statement in the *SQL/MX Reference Manual*.

At this point, program control could return to step described in [Initialize the Dynamic Input Parameters](#) on page 11-5 to continue fetching rows with another input parameter value for the dynamic cursor.

Suppose that you are finished with the dynamic SQL statement that specifies the cursor. Deallocate the resources used by the prepared cursor specification. The module that contains the DEALLOCATE PREPARE statement must also contain a PREPARE statement for *statement-name*.

```
DEALLOCATE PREPARE statement-name
```

For complete syntax, see the DEALLOCATE PREPARE statement in the *SQL/MX Reference Manual*.

Using Date-Time and INTERVAL Data Types

If a column in the select list of a cursor specification has an INTERVAL or standard date-time (DATE, TIME, or TIMESTAMP, or the SQL/MP DATETIME equivalents) data type, use the INTERVAL or date-time type.

If a column in the select list of a cursor specification has a nonstandard SQL/MP DATETIME data type that is not equivalent to DATE, TIME, or TIMESTAMP, you must use the CAST function to convert the column to a character string. You must also specify the length of the target host variable (or the length–1 in the case of a C program) as part of the CAST conversion.

The data type of an input parameter can be either numeric or character. If a column (in the WHERE clause of the cursor specification) to be compared to an input parameter has an INTERVAL or standard date-time data type, the parameter in the USING clause of the OPEN statement must have an INTERVAL or compatible date-time data type. In the WHERE clause, you must cast the parameter to a date-time or INTERVAL data type.

If a column (in the WHERE clause of the cursor specification) to be compared to an input parameter has a nonstandard SQL/MP date-time data type, the parameter in the USING clause of the OPEN statement must have a character data type. In the WHERE clause, you must first specify the data type of the parameter as CHAR in the AS clause of the CAST function, and then cast it to a date-time data type.

Standard Date-Time Example

This example uses a typical context for a standard date-time input parameter for a cursor specification:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4) hv_projcode;
    VARCHAR              hv_projdesc[19];
    DATE                 hv_start_date;
    DATE                 in_start_date;
    char                 curspec[256]; /* Dynamic cursor spec */
EXEC SQL END DECLARE SECTION;

...
strcpy(curspec,
    "SELECT projcode, projdesc, start_date")
```

```

    " FROM samdbcat.persnl.project"
    " WHERE start_date <= CAST(? AS DATE)";

/* Prepare the cursor specification. */
EXEC SQL PREPARE cursor_spec FROM :curspec;

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

/* Initialize the parameter in the WHERE clause. */
printf("Enter the latest 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_by_projcode USING :in_start_date;

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

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    hv_start_date[10]='\0';
    printf("\nProject Code: %hu, Start Date: %s",
        hv_projcode, hv_start_date);
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_by_projcode
        INTO :hv_projcode, :hv_projdesc, :hv_start_date;
}
/* Close the cursor. */
EXEC SQL CLOSE get_by_projcode;

/* Deallocate the prepared cursor specification. */
EXEC SQL DEALLOCATE PREPARE cursor_spec;

```

Interval Example

This example uses a typical context for an interval input parameter for a cursor specification:

COBOL

```

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 sqlstate pic x(5).
    01 hv-projcode pic 9(4) comp.
    01 hv-projdesc pic x(18).
    01 hv-est-complete INTERVAL DAY(3).
    01 curspec pic x(255).
    01 in-est-complete INTERVAL DAY(3).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
    MOVE "SELECT projcode, projdesc, est_complete
-       " FROM samdbcat.persnl.project
-       " WHERE est_complete >=
-       " CAST(? AS INTERVAL DAY(3))"
    TO curspec.
* Prepare cursor specification.
    EXEC SQL PREPARE cursor_spec FROM :curspec END-EXEC.

```

```

* Declare the dynamic cursor from the prepared statement.
  EXEC SQL
    DECLARE get_by_projcode CURSOR FOR cursor_spec
  END-EXEC.

* Initialize the parameter in the WHERE clause.
  DISPLAY "Enter the minimum estimated number of days: ".
  ACCEPT in-est-complete.

* Open the cursor using the values of the dynamic parameter.
  EXEC SQL
    OPEN get_by_projcode USING :in-est-complete
  END-EXEC.

* Fetch the first row of result from table.
  EXEC SQL
    FETCH get_by_projcode
      INTO :hv-projcode, :hv-projdesc, :hv-est-complete
  END-EXEC.

* Fetch rest of the results from table.
  PERFORM UNTIL sqlstate = sqlstate-nodata
    ...
    EXEC SQL FETCH get_by_projcode
      INTO :hv-projcode, :hv-projdesc, :hv-est-complete
    END-EXEC.
  END-PERFORM.

* Close the cursor.
  EXEC SQL CLOSE get_by_projcode END-EXEC.

* Deallocate the prepared cursor specification. */
  EXEC SQL DEALLOCATE PREPARE cursor_spec END-EXEC.

```

Nonstandard SQL/MP DATETIME Example

This example uses a typical context for a nonstandard date-time input parameter, DATETIME MONTH TO DAY (mm-dd), for a cursor specification:

C

```

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  unsigned NUMERIC (4) hv_projcode;
  VARCHAR          hv_projdesc[19];
  char             hv_start_date[6];
  char             in_start_date[6];
  char             curspec[256]; /* Dynamic cursor spec */
EXEC SQL END DECLARE SECTION;

...
strcpy(curspec,
  "SELECT projcode, projdesc, CAST(start_date AS CHAR(5))"
  " FROM samdbcat.persnl.project"
  " WHERE start_date <= "
  "       CAST(CAST(? AS CHAR(5)) AS DATETIME MONTH TO DAY)");

/* Prepare the cursor specification. */
EXEC SQL PREPARE cursor_spec FROM :curspec;

```

```

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

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

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

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

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    hv_start_date[5]='\0';
    printf("\nProject Code: %hu, Start Date: %s",
           hv_projcode, hv_start_date);
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_by_projcode
          INTO :hv_projcode, :hv_projdesc, :hv_start_date;
}
/* Close the cursor. */
EXEC SQL CLOSE get_by_projcode;

/* Deallocate the prepared cursor specification. */
EXEC SQL DEALLOCATE PREPARE cursor_spec;

```

Dynamic SQL Cursors Using Descriptor Areas

When using a dynamic cursor, you can specify an output descriptor area for the FETCH INTO statement instead of a list of host variables. The values of the output parameters are stored in the descriptor area, retrieved by using GET DESCRIPTOR, and assigned to a compatible host variable by testing on the data type. See the *SQL/MX Reference Manual* for information on descriptor areas.

For a complete example of dynamic SQL cursors using descriptor areas, see [Example A-7](#) on page A-18.

12 Dynamic SQL Rowsets

The dynamic SQL statements of NonStop SQL/MX can use rowsets to:

- Provide an array of input values in an INSERT, UPDATE, DELETE, or SELECT statement.
- Retrieve an array of output values in a FETCH statement.

This section describes:

- [Using Dynamic SQL Rowsets](#) on page 12-1
- [Preparing an SQL Statement With Dynamic Rowsets](#) on page 12-2
- [Using the SET DESCRIPTOR Statement](#) on page 12-5
- [Using the GET DESCRIPTOR Statement](#) on page 12-9
- [Using the DESCRIBE INPUT Statement](#) on page 12-10

For a general discussion of rowsets, see [Section 7, Static Rowsets](#).

Using Dynamic SQL Rowsets

You can use of rowsets in dynamic SQL statements as you do rowsets in static SQL statements, with these restrictions:

- Rowset-derived tables are not available from dynamic SQL.
- In dynamic SQL, all input rowsets must be of the same size in an SQL statement. Use of rowsets and scalars for input in the same SQL statement is allowed, as in static SQL, but all rowsets for input must be of the same size in a dynamic SQL statement.

The dynamic SQL programming model is described in:

- [Section 9, Dynamic SQL](#)
- [Section 10, Dynamic SQL With Descriptor Areas](#)
- [Section 11, Dynamic SQL Cursors](#)

To use rowsets in a dynamic SQL statement follow all guidelines described in these sections with these restrictions:

- For a C/C++ application, when dynamic SQL rowsets are used with descriptor areas, data must be exchanged with NonStop SQL/MX using the VARIABLE_POINTER and INDICATOR_POINTER descriptor fields. You cannot use the VARIABLE_DATA and INDICATOR_DATA descriptor fields with dynamic SQL rowsets.
- For a COBOL application, dynamic SQL rowsets with descriptor areas because the VARIABLE_POINTER descriptor field is not supported in COBOL. Dynamic SQL rowsets in COBOL can be accessed only by using dynamic SQL with argument lists, as described in [Steps for Using Dynamic SQL With Argument Lists](#) on page 9-3.

Data can be exchanged between a dynamic embedded SQL application and the database by using rowset-type host variables (as in static SQL) or by using the address of memory locations in application space, which have been suitably prepared to send data to or receive data from the database. Consequently, dynamic rowsets can be used without declaring rowset-type host variables in the DECLARE section. This requires the use of descriptor areas to provide the address of input/output memory locations to NonStop SQL/MX. It is recommended that you use rowset-type host variables to exchange data with the database. The remainder of this section assumes that all input data will be provided in host variables, and output data will be fetched into host variables.

These dynamic SQL statements have modified behavior when rowsets are used.

- PREPARE
- GET DESCRIPTOR
- SET DESCRIPTOR
- DESCRIBE INPUT

For all other types of dynamic SQL statements, the descriptions in [Section 9, Dynamic SQL](#), [Section 10, Dynamic SQL With Descriptor Areas](#), and [Section 11, Dynamic SQL Cursors](#) apply without any change to rowsets. The rowset-specific information of these four statements are described in this section.

Preparing an SQL Statement With Dynamic Rowsets

In dynamic embedded SQL, SQL statements are placed in character strings and are SQL compiled at execution time by calling the PREPARE statement. You might not know the host variables used to transfer data between an application program and a database when the SQL statement is prepared. Input parameters are instead denoted by '?', with an optional CAST specification describing the SQL type of the parameter in the SQL statement that is prepared.

To use dynamic rowsets for input, you must know when calling PREPARE if each input parameter will be executed using a scalar host variable or a rowset host variable. If you use a rowset host variable, you must specify its length (that is, the number of entries in the rowset) in the PREPARE string. In other words, you must indicate that an input parameter is of array type when you specify the SQL statement using the syntax shown in [Specification of an Rowset Parameter in the PREPARE String](#) on page 12-3. You can mix scalar input parameters and array input parameters in the same SQL statement.

To use dynamic rowsets for output, the PREPARE statement is not modified at all from the description in [Section 11, Dynamic SQL Cursors](#). When preparing a cursor specification, you are not required to specify whether you will use rowsets to fetch from a dynamic SQL cursor. You can decide during the execution phase. No prior notice is required during the compile phase. In fact, you can use the same dynamic SQL cursor

to fetch one row at a time and multiple rows at a time by using dynamic rowsets in separate FETCH calls.

Specification of an Rowset Parameter in the PREPARE String

To use a rowset host variable for an input parameter, indicate its length by `'?[positive-integer-constant]'` in the PREPARE string. For example:

```
?[10]
```

The length specifier must be a positive integer (> 0) and a constant. Otherwise, a parse error is raised. The term rowset parameter denotes the parameter represented by `'?[positive-integer-constant]'`. In contrast with static SQL, all rowset parameters used for input in a single SQL statement must be of the same size (for example, N). This approach is logically equivalent to the SQL statement being executed separately N times, with scalar values for input.

To use scalar host variable for an input parameter, use the PREPARE syntax `'?'` as described in [Section 9, Dynamic SQL](#).

As with static SQL, you can mix scalar and array type input parameters in the same SQL statement. Note that with the syntax presented, `?[1]` denotes a rowset parameter of length one, which is distinct from `?`, which denotes a scalar parameter. As explained in [Section 7, Static Rowsets](#), when a scalar input parameter is mixed in with rowset parameters, the same value is used for each of the N logical invocations of the statement. Therefore, `?` implies the same values for each logical invocation, while `?[1]` implies an array with one element. No replication of values exists when `?[1]` is used.

Basic Dynamic Rowset Example

This statement inserts rows into table TAB, 100 rows at a time. All three input host variables are of rowset type with a minimum length of 100. The type of each rowset host variable must be compatible with the type of the column into which its values are inserted.

```
INSERT INTO tab VALUES ([100], [100], [100])
```

Mixing Scalar and Rowset Host Variables Example

This statement inserts rows into table TAB, 50 rows at a time. The first column must be a rowset host variable with a minimum length of 50 and type compatible with the type of the first column. The second column must be a rowset host variable with a minimum length of 50 and type compatible with NUMERIC(4) UNSIGNED. The third column must be a scalar host variable and its type must be compatible with the type of the third column. You can use scalars and rowsets in the same insert list. The same value is used for the scalar at each logical invocation of the statement.

```
INSERT INTO tab VALUES ([50] , CAST([50] AS NUMERIC(4)
UNSIGNED) , ?)
```

Using the FOR INPUT SIZE and KEY BY Syntax Example

This statement uses the FOR INPUT SIZE and KEY BY syntax with an array input parameter in the WHERE clause. The input size is always a scalar parameter, and its value must be less than the length of the input parameter array.

```
ROWSET FOR INPUT SIZE ? KEY BY rowid SELECT rowid, COUNT(*),  
a FROM tab WHERE b = ?[50] GROUP BY rowid, a
```

Matching Compile-Time Specified Length With Execution-Time Length

When you specify rowset type host variables during execution to take the place of parameter arrays specified in the PREPARE string, their length must match the integer constant provided for that parameter array in the PREPARE string, if an INPUT SIZE clause is not present in the SQL statement.

Use rowset host variables of sufficient length. NonStop SQL/MX might raise an error during execution of the statement if the size of an input rowset host variable provided is less than N , the common size of all input rowset parameters. The error is raised during execution for reading from or writing into memory that was not intended for that purpose. In other words, NonStop SQL/MX does not perform a bounds check on rowset host variables used during execution. Memory access violations are reported at execution time if these guidelines are not followed. Also, memory might be overwritten or falsely read from, based on the specific circumstance. Memory access violations occur when an input rowset host variable is less than size N . You cannot assume that these errors will be raised by NonStop SQL/MX during execution time.

You can compile with rowset parameters and use scalar values at execution time if you can use at least one rowset host variable during execution. The scalar value is duplicated as many times as specified in the PREPARE string.

An error will be raised during execution if:

- You specify one or more parameter arrays in the PREPARE string, and all host variables used at run time for input are scalars.
- You specify a scalar parameter (not of array type) in the PREPARE string and use a rowset host variable to transfer data for that parameter.

When you specify more than one rowset parameter in the PREPARE string, at most all but one of the rowset parameters can be provided scalar values at run time. The rowset parameters that have scalar values provided for them at execution time behave as if the query were compiled with scalar parameters in this location. The scalar value is duplicated by NonStop SQL/MX as many times as the size of the input array value.

Note that when you use rowset host variables of size larger than N for input, values beyond the N th entry are ignored.

Dynamic SQL With Argument Lists

If you use dynamic SQL with argument lists only and not descriptor areas, see the discussion [Preparing an SQL Statement With Dynamic Rowsets](#) on page 12-2. The next subsections apply only to dynamic SQL with descriptor areas. Rowset host variables can also be used in the USING and INTO clauses of the EXECUTE statement for input and output variables, respectively. In this case, you do not need to set any descriptor fields.

Using the SET DESCRIPTOR Statement

Note. Dynamic SQL rowsets with descriptor areas are not supported in COBOL.

Use dynamic SQL rowsets with descriptor areas when you need to use descriptor areas, such as when the number of arguments is not known until execution time. Follow the procedures described in [Section 10, Dynamic SQL With Descriptor Areas](#). In addition, these practices are unique to dynamic SQL rowsets.

- You must set the appropriate value in the rowset-related descriptor fields, ROWSET_SIZE, ROWSET_VAR_LAYOUT_SIZE and ROWSET_IND_LAYOUT_SIZE before execution.

For an input descriptor, use the DESCRIBE statement after PREPARE to set the appropriate values set in the input descriptor.

For an output descriptor, set all three descriptor fields, even if the DESCRIBE OUTPUT statement is used. The DESCRIBE statement uses compile-time data to populate the descriptor and, when using dynamic rowsets as output, no rowset specific information exists in the statement that was prepared (see [Preparing an SQL Statement With Dynamic Rowsets](#) on page 12-2). Use of dynamic rowsets for output is an execution-time decision and cannot be inferred by DESCRIBE.

- When you use rowset host variables to transfer data to and from a database, VARIABLE_DATA and INDICATOR_DATA descriptor item fields must not be used. Data can be exchanged between an application and a database only if you use the VARIABLE_POINTER and INDICATOR_POINTER descriptor fields. Set the VARIABLE_POINTER and INDICATOR_POINTER fields to appropriate values before executing the statement.

Setting the Rowset-Specific Descriptor Fields

You set the rowset-related descriptor fields ROWSET_SIZE, ROWSET_VAR_LAYOUT_SIZE and ROWSET_IND_LAYOUT_SIZE with the SET DESCRIPTOR statement. The syntax is:

```
SET DESCRIPTOR descriptor-name set-descriptor-info
```

descriptor-name is:

```
[GLOBAL | LOCAL] value-specification
```

```

set-descriptor-info is:
COUNT = value-specification
| ROWSET_SIZE = value-specification
| VALUE item-number set-item-info [, set-item-info]...

```

```

set-item-info is:
descriptor-item-name = value-specification

```

```

descriptor-item-name is:
ROWSET_VAR_LAYOUT_SIZE
| ROWSET_IND_LAYOUT_SIZE
| other-descriptor-item-names

```

The *value-specification* can be a literal or a host variable with exact numeric data type. For the full description of the SET DESCRIPTOR statement, including the *other-descriptor-item-names*, see the *SQL/MX Reference Manual*.

ROWSET_SIZE

Use the ROWSET_SIZE descriptor field header to specify the number of rows to be transmitted to and from a database while executing SQL statements. For input descriptors, this field must contain the value *N* as described in [Specification of an Rowset Parameter in the PREPARE String](#) on page 12-3. For output descriptors, this field must contain the common length (for example, *M*) of all rowset output variables that you will use to retrieve data from the dynamic SQL cursor. Multiple FETCH statements can be issued on a cursor, and the value of ROWSET_SIZE can be different for each FETCH call. This value is equal to the maximum number of rows that will be retrieved by the FETCH statement.

This field occurs only once per descriptor regardless of how many individual items might be contained in that descriptor. A descriptor does not contain a field to store the size of each individual rowset host variable used to transfer data. Instead, a descriptor contains one field to store the common rowset size for all input or output arrays. For input descriptors, this value denotes *N*, the number of times the SQL statement is logically executed with separate scalar input value sets. However, if there is a ROWSET FOR INPUT SIZE clause in the SQL statement, the value provided for the INPUT SIZE must be less than or equal to *N*, and the statement is only logically executed INPUT SIZE number of times.

For input descriptors, DESCRIBE can be used to set the value of this field to *N*. For output descriptors, set this field manually.

ROWSET_VAR_LAYOUT_SIZE

Use the ROWSET_VAR_LAYOUT_SIZE item field in a descriptor to specify the size of an individual array element in a rowset host variable. A value 0 (zero) in this field denotes that the host variable is not of rowset type and is a scalar host variable. For rowset host variables, this field is equal to the size of an individual array element. If you

set this descriptor field manually, use the `sizeof` function and make its value equal to `sizeof (individual array element)`.

Note. The SQL/MX extension TYPE -601 (character varying with length specified in the first two bytes) is a special case. If rowsets are to be used with this data type, the ROWSET_VAR_LAYOUT_SIZE field must not include the two bytes used to specify varying character length. In this case, the ROWSET_VAR_LAYOUT_SIZE field must contain the length of the data buffer from the end of one two-byte length specification to the start of the next two-byte length specification, in bytes (that is, step size for the data buffer that contains the character varying data). Use a struct with two fields to hold the two-byte length specification in one field and the character data in the second field. If you do not, then the space allocated for each character value (maximum length of varchar) must be an even number. If you use structs to hold the length specification and varchar data, the maximum length of varchar can be either even or odd.

[Table 12-1](#) lists minimum values you can use to set the ROWSET_VAR_LAYOUT_SIZE descriptor field for various data types. Values larger than those listed do not raise an error provided the individual array element is also equally large.

Table 12-1. Minimum Values for ROWSET_VAR_LAYOUT_SIZE Descriptor Field (page 1 of 2)

SQL Data Type	ROWSET_VAR_LAYOUT_SIZE
CHAR[ACTER](1)	(1 + 1) if CHARSET = ISO88591
PIC[TURE] X(1)	(1 + 1) * 2 if CHARSET = UCS2, KANJI OR KSC5601 *
CHAR[ACTER] VARYING (1)	
VARCHAR[ACTER] (1)	
NUMERIC (1 to 4, s) SIGNED	2
NUMERIC (1 to 4, s) UNSIGNED	
NUMERIC (5 to 9, s) SIGNED	4
NUMERIC (5 to 9, s) UNSIGNED	
NUMERIC (10 to 18, s) SIGNED	8
NUMERIC (10 to 18, s) UNSIGNED	
PIC[TURE] [S] 9(1-s)V9(s) COMP	Same as NUMERIC
DEC[IMAL] (1, s) SIGNED	1+2 if TYPE_FS=151 (_SQLDT_DES_LSS) 1+1 if TYPE_FS=152 (_SQLDT_DES_LSE)
DEC[IMAL] (1, s) UNSIGNED	1+1
PIC[TURE] [S] 9(1-s)V9(s)	Same as DECIMAL UNSIGNED
SMALLINT SIGNED	2
SMALLINT UNSIGNED	

* All character format data is assumed to be null terminated. If character format data, such as CHAR, VARCHAR, DECIMAL, DATE, TIME, TIMESTAMP, or INTERVAL is not null terminated, subtract the null terminator bytes from this table. For double-byte character sets, multiply the length of the string (in characters) by 2 to get the length in bytes.

** See on page 3-35 for the appropriate value of 1 for TIME/TIMESTAMP precision.

*** See [INTERVAL Representation](#) on page 3-38 for guidance in computing 1. The sign byte is included in 1, and the extra byte (+1) is for the null terminator.

Table 12-1. Minimum Values for ROWSET_VAR_LAYOUT_SIZE Descriptor Field (page 2 of 2)

SQL Data Type	ROWSET_VAR_LAYOUT_SIZE
INT[EGER] SIGNED	4
INT[EGER] UNSIGNED	
LARGEINT	8
FLOAT (1 to 22 bits)	4
REAL	
FLOAT (23 to 54 bits)	8
DOUBLE PRECISION	
DATE	11
TIME, TIMESTAMP	1+1 **
INTERVAL	1+1 ***

* All character format data is assumed to be null terminated. If character format data, such as CHAR, VARCHAR, DECIMAL, DATE, TIME, TIMESTAMP, or INTERVAL is not null terminated, subtract the null terminator bytes from this table. For double-byte character sets, multiply the length of the string (in characters) by 2 to get the length in bytes.

** See on page 3-35 for the appropriate value of 1 for TIME/TIMESTAMP precision.

*** See [INTERVAL Representation](#) on page 3-38 for guidance in computing 1. The sign byte is included in 1, and the extra byte (+1) is for the null terminator.

For input descriptors, use DESCRIBE to set the value of this field to the value indicated in [Table 12-1](#). For DECIMAL and PICTURE 9 DISPLAY data types, the value provided by DESCRIBE for this field is one less than the value in [Table 12-1](#). The first byte to store the sign is not accounted for by DESCRIBE because DESCRIBE expects DECIMAL data to be provided in Leading Sign Embedded (LSE, with TYPE_FS = 152) format. If the Leading Sign Separate (LSS, with TYPE_FS = 151) format is used for input, one must be added to the value of ROWSET_VAR_LAYOUT_SIZE provided by DESCRIBE. Set this field manually for output descriptors.

If the value specified for this descriptor item field does not follow the rules described here, an execution-time error might be raised for data type mismatch. Memory access violations might occur if the value for this descriptor field does not follow the rules described here.

ROWSET_IND_LAYOUT_SIZE

The ROWSET_IND_LAYOUT_SIZE is an item field in a descriptor that specifies the size of an individual array element in a rowset host variable used as an INDICATOR. A value 0 (zero) in this field denotes that the indicator host variable is not of rowset type and is a scalar type. For indicator host variables, ROWSET_IND_LAYOUT_SIZE denotes the length in bytes for the exact numeric data type used.

For input descriptors, use DESCRIBE to set the value of this field to be equal to two. DESCRIBE assumes that indicator data provided is of type `signed short`. If another type such as `signed long` is used, you must manually set the value of this descriptor field. For output descriptors, you also must set this field manually.

If the value specified for this descriptor item field does not meet these guidelines, an execution time error of memory access violation or data type mismatch can result.

Exclusive Use of VARIABLE_POINTER and INDICATOR_POINTER

When you use rowset host variables to transfer data to and from a database, VARIABLE_DATA and INDICATOR_DATA descriptor item fields must not be used. Data can be exchanged between an application and a database only by using the VARIABLE_POINTER and INDICATOR_POINTER descriptor fields.

For information on using the VARIABLE_POINTER and INDICATOR_POINTER descriptor fields, see [Section 10, Dynamic SQL With Descriptor Areas](#).

Set the ROWSET_VAR_LAYOUT_SIZE field before setting the VARIABLE_POINTER field to the address of a rowset host variable. Similarly for indicators, set the ROWSET_IND_LAYOUT_SIZE field before setting the INDICATOR_POINTER field to the address of a rowset indicator host variable. An error occurs if the VARIABLE_DATA or INDICATOR_DATA descriptor item fields are used after the ROWSET_VAR_LAYOUT_SIZE or ROWSET_IND_LAYOUT_SIZE fields have been set to nonzero values.

Using the GET DESCRIPTOR Statement

Use the GET DESCRIPTOR statement to obtain the value of rowset specific descriptor fields, described in [Using the SET DESCRIPTOR Statement](#) on page 12-5. This strategy is useful after a DESCRIBE INPUT statement has been issued, and you want to determine the value set by DESCRIBE for these descriptor fields. You can also use this statement to check a previously issued SET DESCRIPTOR statement.

The syntax is:

```
GET DESCRIPTOR descriptor-name get-descriptor-info
```

descriptor-name is:

```
[GLOBAL | LOCAL] value-specification
```

get-descriptor-info is:

```
variable-name = COUNT  
| variable-name = ROWSET_SIZE  
| VALUE item-number get-item-info [, get-item-info]...
```

get-item-info is:

```
variable-name = descriptor-item-name
```

descriptor-item-name is:

```
ROWSET_VAR_LAYOUT_SIZE  
| ROWSET_IND_LAYOUT_SIZE  
| other-descriptor-item-names
```

For the full description of the GET DESCRIPTOR Statement, including the *other-descriptor-item-names*, see the *SQL/MX Reference Manual*. The *variable-name* must be a host variable with exact numeric data type.

Using the DESCRIBE INPUT Statement

Use the DESCRIBE INPUT statement after PREPARE to fill rowset-specific descriptor fields with appropriate values. This strategy frees you from setting these descriptor fields manually before execution. However, because DESCRIBE INPUT records compile-time information only, you must manually set descriptor fields if the value provided by DESCRIBE is not appropriate for the execution-time environment in the application.

For the ROWSET_SIZE descriptor header field, DESCRIBE INPUT sets the value of this field to *N*, the common size of all input rowset parameters in the SQL statement. This field is set to *N* even if there are multiple scalar parameters in the SQL statement and only one rowset parameter for input exists. If no rowset parameters for input exist, this field is set to zero by DESCRIBE INPUT.

For the ROWSET_VAR_LAYOUT_SIZE descriptor item field, DESCRIBE INPUT sets its value to the value of OCTET_LENGTH + NT, where OCTET_LENGTH is an item descriptor field, as defined in the *SQL/MX Reference Manual*, if an rowset parameter is specified for input or to zero otherwise.

The variable NT is defined as the size of the null terminator:

- NT is zero for all numeric data types except DECIMAL.
- NT is one for all fixed-length character data types and ANSI VARCHAR, DATE, TIME, TIMESTAMP, and INTERVAL data types if a single-byte character set is used.
- NT is two for all fixed-length character data types and ANSI varchar if a double-byte character set is used. For the SQL/MX extension TYPE -601 (character varying with length specified in the first two bytes), NT equals zero. Note that for this type, the value specified by DESCRIBE for ROWSET_VAR_LAYOUT_SIZE does not include the two bytes used for length specification.

If the individual array element size of the rowset host variable used is of a different size, the value provided by DESCRIBE INPUT must be overwritten.

For the ROWSET_IND_LAYOUT_SIZE descriptor item field, DESCRIBE INPUT sets the value of this field to two, if an rowset parameter was specified for input, or to zero, otherwise. If indicator data is provided in a type other than signed short, then the value provided by DESCRIBE INPUT must be overwritten.

DESCRIBE OUTPUT does not provide information on rowset-specific descriptor fields. If rowsets are used for output, the value provided by DESCRIBE OUTPUT must be overwritten.

Exception Handling and Error Conditions

In NonStop SQL/MX, a warning or error condition is also referred to as an *exception*. Your host language application program can detect these exceptions and can gather diagnostic information after the execution of each of the program's SQL statements. To properly account for various error scenarios, you need to be aware of the variety of exceptions that exist and some of the options that you can use to handle them:

- You can check the value of an SQLSTATE variable and then use the GET DIAGNOSTIC statement to obtain SQL/MX-specific exception information after each SQL statement.
- You can use the WHENEVER statement to map a particular action or response to an error or warning each time that error or warning occurs.

This section provides information for detecting and handling various types of exceptions.

Note. To write your application to recover from temporary network or hardware service interruptions, see the *SQL/MX Release 3.2 Management Manual*.

This section describes:

- [Checking the SQLSTATE Variable](#) on page 13-1
- [Checking the SQLCODE Variable](#) on page 13-5
- [SQL/MX Exception Condition Messages](#) on page 13-11
- [Using the WHENEVER Statement](#) on page 13-13
- [Accessing and Using the Diagnostics Area](#) on page 13-17
- [Special SQL/MX Error Conditions](#) on page 13-20

Checking the SQLSTATE Variable

After the execution of each SQL statement, NonStop SQL/MX returns a value to the SQLSTATE variable to indicate the results of the statement. A host language program can use conditional statements to check the SQLSTATE value to determine the results of execution.

While NonStop SQL/MX does support the SQLCODE variable, use the SQLSTATE variable to detect exception conditions. For equivalent values, see the SQLSTATE values returned by NonStop SQL/MX in the *SQL/MX Messages Manual*.

Declaring SQLSTATE

In a C program, declare SQLSTATE as a *char* array of 6 bytes (*char[6]*), within the Declare section. In a COBOL program, declare SQLSTATE of type PIC (5) within the Declare section.

Examples

In a C program, you must include an extra character for the null terminator. Declare SQLSTATE within the Declare section:

C

```
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    ...
EXEC SQL END DECLARE SECTION;
```

In a COBOL program, declare SQLSTATE within the Declare Section:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 SQLSTATE PIC X(5).
    ...
EXEC SQL END DECLARE SECTION END-EXEC.
```

SQL:1999 SQLSTATE Values

The SQLSTATE variable is a five-character string with two parts. The first part is a two-character class code, and the second part is a three-character subclass code. An SQLSTATE value of 00000 indicates successful completion. In C programs, declare SQLSTATE as type *CHAR*, with a length of six characters to allow for the null terminator.

[Table 13-1](#) lists the SQL:1999 SQLSTATE class and subclass values that are implemented in NonStop SQL/MX.

Table 13-1. SQL:1999 SQLSTATE Class and Subclass Values (page 1 of 2)

Condition	Class	Subcondition	Subclass
Successful completion	00	(No subclass)	000
Warning	01	(No subclass)	000
		String data, right truncation	004
		Privilege not revoked	006
		Privilege not granted	007
No data	02	(No subclass)	000
Dynamic SQL error	07	(No subclass)	000
		Invalid descriptor count	008
		Invalid descriptor index	009
Feature not supported	0A	(No subclass)	000
Cardinality violation	21	(No subclass)	000

Table 13-1. SQL:1999 SQLSTATE Class and Subclass Values (page 2 of 2)

Condition	Class	Subcondition	Subclass
Data exception	22	(No subclass)	000
		Null, no indicator parameter	002
		Numeric value out of range	003
		Invalid date-time format	007
		Date-time field overflow	008
		Substring error	011
		Division by zero	012
		Invalid escape character	019
		Unterminated C string	024
		Invalid escape sequence	025
		Trim error	027
Integrity constraint violation	23	(No subclass)	000
Invalid cursor state	24	(No subclass)	000
Invalid transaction state	25	(No subclass)	000
Invalid SQL statement name	26	(No subclass)	000
Invalid authorization specification	28	(No subclass)	000
Dependent privilege descriptors still exist	2B	(No subclass)	000
Invalid character set name	2C	(No subclass)	000
Invalid SQL descriptor name	33	(No subclass)	000
Invalid catalog name	3D	(No subclass)	000
Invalid schema name	3F	(No subclass)	000
Transaction rollback	40	(No subclass)	000
		Statement completion unknown	003
Syntax error or access rule violation	42	(No subclass)	000
With check option violation	44	(No subclass)	000

SQL/MX SQLSTATE Values

NonStop SQL/MX has extended SQL:1999 SQLSTATE values to include other situations not described by ANSI values. If an SQL:1999 SQLSTATE value exists for an error condition, NonStop SQL/MX returns that value, as listed in [Table 13-1](#) on page 13-2. Otherwise, NonStop SQL/MX returns a value listed in [Table 13-2](#).

Table 13-2. Mapping of SQLCODE to SQL/MX-Defined SQLSTATE Values

SQLCODE	SQLSTATE	Class Origin	Subclass Origin	Description
0	00000	ISO 9075	ISO 9075	Successful completion
100	02000	ISO 9075	ISO 9075	No data
n < 0	X0yzz	SQL/MX	SQL/MX	Error
n > 0 (<> 100)	01yzz	ISO 9075	SQL/MX	Warning

In [Table 13-2](#), for the last two cases, the subclass abbreviated yzz is in one of the following ranges: W00 through W09, W0A through WZZ, X00 through X09, or X0A through XZZ.

Using SQLSTATE

After you declare SQLSTATE within a Declare Section, use conditional statements to check the SQLSTATE variable after the execution of an SQL statement.

This example checks the SQLSTATE variable for errors or warnings after an UPDATE statement. SQLSTATE is declared as global so that it can be referenced in the `process_sqlstate` function:

Example

C

```
void process_sqlstate(void);

EXEC SQL BEGIN DECLARE SECTION;
  char SQLSTATE[6];
  ...
EXEC SQL END DECLARE SECTION;
...
int main()
{
  char SQLSTATE_OK[6]="00000";
  SQLSTATE[5]='\0';
  SQLSTATE_OK[5]='\0';
  ...
  EXEC SQL BEGIN WORK;
  EXEC SQL UPDATE customer SET CREDIT = 'CR';
  if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    printf ("\nRows were updated!");
    EXEC SQL COMMIT WORK;
  } else process_sqlstate();
  ...
  return 0;
} /* end main */

void process_sqlstate(void)
{
  printf ("\nError or warning occurred! SQLSTATE = %s",SQLSTATE);
```

```
... /* Process the SQL error. */
} /* end process_sqlstate */
```

This example checks the value of the SQLSTATE variable only after the UPDATE statement. To ensure your program is executing properly, you must check SQLSTATE after every SQL statement. For further information on how to do this without error checking after every statement, see [Using the WHENEVER Statement](#) on page 13-13.

Example

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE          PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.
01 SQLSTATE-OK       PIC X(5) VALUE "00000".
...
EXEC SQL BEGIN WORK END-EXEC.
EXEC SQL UPDATE customer SET CREDIT = 'CR' END-EXEC.
IF SQLSTATE = SQLSTATE-OK
    DISPLAY "Rows were updated!"
    EXEC SQL COMMIT WORK END-EXEC.
ELSE PERFORM 1000-PROCESS-SQLSTATE.
...
1000-PROCESS-SQLSTATE.
    DISPLAY "Error or warning occurred! SQLSTATE = " SQLSTATE.
* Process the SQL error
...
```

Checking the SQLCODE Variable

Although checking the SQLSTATE variable is recommended to detect exception conditions, NonStop SQL/MX also supports the SQLCODE variable.

Declaring SQLCODE

In a C program, declare SQLCODE as an integer variable of type `long` within the scope of each embedded SQL statement in your program.

In a COBOL program, declare SQLCODE as an integer variable of type `S9(9) COMP`.

Declaring SQLCODE and SQLSTATE

If you want to use both SQLCODE and SQLSTATE in your program, you must declare them within the Declare section, with the correct data type. By default, preprocessor does not return any error if SQLCODE is declared incorrectly. However, if SQLSTATE is not declared correctly, preprocessor returns an error.

The `-w` option is used to detect the missing or incorrect SQLCODE and SQLSTATE variables. The following arguments are supported for the `-w` option:

- `Sqlcode`
- `Sqlstate`

- Both

If the SQLCODE or SQLSTATE is missing or incorrect, the following behavior is observed:

- If SQLCODE or SQLSTATE is missing or incorrect, the preprocessor will issue a warning for SQLCODE, SQLSTATE, or both depending on the argument for the `-w` option.
- If the SQLCODE or SQLSTATE is missing in a scope, warning 13085 will be issued when the first embedded SQL statement within the scope is encountered.
- If the SQLCODE or SQLSTATE is declared incorrectly within the Declare section, warning 13086 will be issued at the end of the SQL Declare section.
- If the `-w` option is not specified, the default behavior is expected.

For information on how to use the `-w` option, see [Syntax for the OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-20 and [Syntax for the Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-28.

[Table 13-3](#) describes the behavior of preprocessor, embedded application, and language compiler for SQLCODE and SQLSTATE with different scopes. It is assumed that SQLSTATE and SQLCODE are declared correctly and are used to handle exception conditions.

Table 13-3. SQLCODE and SQLSTATE missing declaration

SQLCODE	SQLSTATE	Values of <code>-w</code> command line option for the preprocessor			Language Compiler	Values returned to the application		default
		sqlcode	sqlstate	Both		SQLCODE	SQLSTATE	
IN	IN	No warning	No warning	No warning	No Error	Yes	Yes	No Warning
IN	OUT	No warning	Warning for SQLSTATE	Warning for SQLSTATE	No Error	Yes	No	No Warning
IN	ND	No warning	Warning for SQLSTATE	Warning for SQLSTATE	Error for SQLSTATE	Yes	No	No Warning
OUT	IN	Warning for SQLCODE	No warning	Warning for SQLCODE	No Error	No	Yes	No Warning
OUT	OUT	Warning for SQLCODE	Warning for SQLSTATE	Warning for both	No Error	Yes	No	No Warning

Table 13-3. SQLCODE and SQLSTATE missing declaration

SQLCODE	SQLSTATE	Values of <code>-w</code> command line option for the preprocessor			Language Compiler	Values returned to the application		default
		sqlcode	sqlstate	Both		SQLCODE	SQLSTATE	
OUT	ND	Warning for SQLCODE	Warning for SQLSTATE	Warning for both	Error for SQLSTATE	Yes	No	No Warning
ND	IN	Warning for SQLCODE	No warning	Warning for SQLCODE	Error for SQLCODE	No	Yes	No Warning
ND	OUT	Warning for SQLCODE	Warning for SQLSTATE	Warning for both	Error for SQLCODE	No	No	No Warning
ND	ND	Warning for SQLCODE	Warning for SQLSTATE	Warning for both	Error for both	No	No	No Warning

IN - Inside the Declare section

OUT - Outside the Declare section

ND - Not declared in the application

[Table 13-4](#) describes the behavior of preprocessor and embedded application for incorrect SQLCODE or SQLSTATE or both.

Table 13-4. SQLCODE and SQLSTATE incorrect declaration

SQLCODE	SQLSTATE	Values of <code>-w</code> command line option for the preprocessor			default	Values returned to the application	
		sqlcode	sqlstate	Both		SQLCODE	SQLSTATE
Correct	Correct	No warning	No warning	No warning	No warning	Yes	Yes

Table 13-4. SQLCODE and SQLSTATE incorrect declaration

SQLCODE	SQLSTATE	Values of -w command line option for the preprocessor			default	Values returned to the application	
		sqlcode	sqlstate	Both		SQLCODE	SQLSTATE
Wrong	Correct	Warning for SQLCODE	No warning	Warning for SQLCODE	No warning	No	Yes
Correct	Wrong	No warning	Warning for SQLSTATE	Warning for SQLSTATE	Error for SQLSTATE	Yes	No
Wrong	Wrong	Warning for SQLCODE	Warning for SQLSTATE	Warning for both	Error for SQLSTATE	No	No

Correct - SQLCODE declared as long SQLCODE; SQLSTATE declared as char SQLSTATE[6].

Wrong - SQLCODE and SQLSTATE declared as any other data type.

The following embedded application `a.sql` is used to explain the behavior of the `-w` option with various arguments:

```
#include <stdio.h>
void Func1(void);
int main()
{
    long SQLCODE;
    //outside declare section
    EXEC SQL BEGIN DECLARE SECTION;
    int SQLSTATE[6];
    //inside declare section
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INSERT INTO tt1 VALUES(10,20);
    Func1();
}
Void Func1()
{
    EXEC SQL INSERT INTO tt1 VALUES(11,21);17
}
```

The preprocessor returns the following warnings, depending upon the argument specified for the `-w` option:

- `mxsqlc a.sql -c a.cpp -m a.mdf -w sqlcode`

```
*** WARNING[13085] SQLCODE is not declared inside the Declare
Section.
*** WARNING[13025] Warning(s) near line 11.
*** WARNING[13085] SQLCODE is not declared inside the Declare
Section.
*** WARNING[13025] Warning(s) near line 16.
```
- `mxsqlc a.sql -c a.cpp -m a.mdf -w SQLstate`

```
*** WARNING[13086] SQLSTATE is not declared of type char[6]
inside the Declare Section.
*** WARNING[13025] Warning(s) near line 8.
*** WARNING[13085] SQLSTATE is not declared inside the
Declare Section.
*** WARNING[13025] Warning(s) near line 16.
```

- `mssqlc a.sql -c a.cpp -m a.mdf -w both`

```
*** WARNING[13086] SQLSTATE is not declared of type char[6]
inside the Declare Section.
*** WARNING[13025] Warning(s) near line 8.
*** WARNING[13085] SQLCODE is not declared inside the Declare
Section.
*** WARNING[13025] Warning(s) near line 11.
*** WARNING[13085] SQLCODE is not declared inside the Declare
Section.
*** WARNING[13085] SQLSTATE is not declared inside the
Declare Section.
*** WARNING[13025] Warning(s) near line 16.
```

SQLCODE Values

After the execution of an embedded SQL statement, NonStop SQL/MX returns the values listed in [Table 13-5](#) to SQLCODE. The SQL message numbers described under [Using the WHENEVER Statement](#) on page 13-13 are SQLCODE values except for the special values 0 and 100. The SQL message numbers that indicate errors are stored as negative numbers, and the SQL message numbers that indicate warnings are stored as positive numbers.

Table 13-5. SQLCODE Values

Value	Status
< 0	An error occurred.
> 0 (<>100)	A warning occurred.
100	No data was found.
0	The statement completed successfully.

Using SQLCODE

You can declare SQLCODE as a global variable at the start of each source unit that contains embedded SQL statements. You can then use conditional statements to check the SQLCODE variable after the execution of an SQL statement.

This example checks the SQLCODE variable for any errors or warnings after an INSERT statement:

Examples

```
...
/* global C declarations */
long SQLCODE;          /* Declare SQLCODE with data type long. */
...
/* Set new_jobcode and new_jobdesc host variables. */
```

```
EXEC SQL INSERT INTO sales.job (jobcode,jobdesc)
      VALUES (:new_jobcode,:new_jobdesc);

if (SQLCODE == 0) { printf ("\nValues were inserted!");
      EXEC SQL COMMIT WORK; }
else process_sqlcode();
```

Note. You must declare `SQLCODE` anywhere in your program or declare `SQLSTATE` within a `Declare` section. If you do not verify `SQLSTATE`, you must not declare `SQLSTATE` because query execution requires string processing to return the `SQLSTATE` value. However, because verifying `SQLCODE` is a deprecated operation, HP recommends that you verify `SQLSTATE` as the means to detect exception conditions and to avoid compiler-generated warnings.

...

```
void process_sqlcode(void)
{
  printf("\nError or warning occurred! SQLCODE = %d",SQLCODE);
  ... /* Process the SQL error. */
}
```

COBOL

```
* Global COBOL declarations
* Declare SQLCODE with data type PIC S9(9) COMP.
01 SQLCODE PIC S9(9) COMP.
...
* Set new_jobcode and new_jobdesc host variables.
EXEC SQL INSERT INTO sales.job (jobcode,jobdesc)
      VALUES (:new-jobcode,:new-jobdesc)
END-EXEC.

IF SQLCODE = 0 DISPLAY "Values were inserted!"
  EXEC SQL COMMIT WORK END-EXEC.
ELSE PERFORM 1000-PROCESS-SQLCODE.
...
1000-PROCESS-SQLCODE.
  DISPLAY "Error or warning occurred! SQLCODE = " SQLCODE.
* Process the SQL error
...
```

SQL/MX Exception Condition Messages

NonStop SQL/MX reports exception condition messages within the MXCI, in SQL preprocessor and compiler listings, and during the execution of embedded SQL programs. You can obtain the messages for an SQL statement within a program by accessing the SQL diagnostics area. For information on each message, see the *SQL/MX Messages Manual*.

Viewing the SQL Messages

To view a list of all SQL messages, see the appropriate messages manual.

The message key is a sequential SQL/MX message number that is returned automatically by NonStop SQL/MX when an error condition occurs. For example, this error message might be displayed within your application development tool while it is preparing an embedded SQL program:

```
*** ERROR[1000] A syntax error occurred.
```

This message number is the SQLCODE value (without the sign). You can view the message for ERROR[1000]:

SQL 1000

1000 42000 A syntax error occurred.

Cause. Syntax was entered incorrectly.

Effect. SQL is unable to prepare the statement.

Recovery. Correct the syntax error and resubmit.

The second number, if present, is the corresponding SQL:1999 SQLSTATE value. In this example, SQLSTATE 42000 is an ANSI value.

Within MXCI, you can display text associated with a message number or SQLCODE value by using the ERROR command. For further information, see the ERROR Command in the *SQL/MX Reference Manual*.

Accessing SQL Messages Within a Program

To obtain error messages that result from the execution of an SQL statement within a program, use the GET DIAGNOSTICS statement to access the SQL diagnostics area.

For example, you might code your program to display:

```
SQLSTATE: 22001
SQLCODE : -8402
Message : *** ERROR[8402] A string overflow occurred during
          the evaluation of a character expression.
```

The corresponding fields in the diagnostics area are RETURNED_SQLSTATE, SQLCODE, and MESSAGE_TEXT, respectively. The message text also provides the SQL message number.

See [Accessing and Using the Diagnostics Area](#) on page 13-17.

Using the WHENEVER Statement

The WHENEVER declaration specifies an action that a program takes, depending on the results of subsequent SQL statements. When you specify WHENEVER, the SQL/MX preprocessor generates statements in your program that perform run-time checking using the SQLSTATE variable after each SQL statement executes.

The generated statements check for these conditions:

- NOT FOUND condition: No data was found. SQLSTATE is 02000, and SQLCODE is 100.
- SQLERROR condition: An SQL error occurred. SQLSTATE indicates an exception condition as shown in [Checking the SQLSTATE Variable](#) on page 13-1. SQLCODE is less than zero.
- SQL_WARNING condition: An SQL warning occurred. SQLSTATE does not indicate a no-data or an error condition. SQLCODE is greater than zero and not equal to 100.

You must specify the WHENEVER declaration in your program before the SQL statements to which it applies. Use this general syntax:

```
WHENEVER { NOT FOUND | SQLERROR | SQL_WARNING }

      {
        CONTINUE
      | GOTO host-label-identifier
      | CALL C-function
      | PERFORM COBOL-routine      }
```

For complete syntax, see the WHENEVER Declaration in the *SQL/MX Reference Manual*.

This example uses WHENEVER declarations to check for the NOTFOUND, SQLERROR, and SQL_WARNING conditions.

Examples

C

```
/* global C declarations */
...
EXEC SQL WHENEVER NOT FOUND GOTO data_not_found;
EXEC SQL WHENEVER SQLERROR GOTO end_prog;
EXEC SQL WHENEVER SQL_WARNING CONTINUE;
EXEC SQL WHENEVER NOT FOUND CALL handle_nodata;
...
```

COBOL

```
* global COBOL declarations
...
EXEC SQL WHENEVER NOT FOUND GOTO data-not-found END-EXEC.
EXEC SQL WHENEVER SQLERROR GOTO end_prog END-EXEC.
EXEC SQL WHENEVER SQL_WARNING CONTINUE END-EXEC.
EXEC SQL WHENEVER NOT FOUND PERFORM handle-nodata END-EXEC.
...
```

Precedence of Multiple WHENEVER Declarations

When more than one WHENEVER declaration applies to an SQL statement, NonStop SQL/MX processes the conditions in order of precedence:

1. NOT FOUND
2. SQLERROR
3. SQL_WARNING

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.

Determining the Scope of a WHENEVER Declaration

The order in which WHENEVER declarations appear in the listing determines their scope:

- A WHENEVER declaration remains in effect until another WHENEVER declaration for the same condition appears. If you want to execute a different routine when an error occurs, specify a new WHENEVER declaration with a different CALL routine.

For example, in a C program, to insert a new row when a row is not found, specify a new WHENEVER declaration:

```
EXEC SQL WHENEVER NOT FOUND CALL insert_row;
```

The new WHENEVER declaration remains in effect until it is disabled or changed.

- If a WHENEVER declaration is coded in a function, the declaration 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 declaration to remain in effect, disable it at the end of the function described next in [Enabling and Disabling the WHENEVER Declaration](#).
- A WHENEVER declaration does not affect SQL statements if they appear in the program before WHENEVER.
- If you are debugging a program and you use a WHENEVER declaration to call an error handling procedure, you might need to save the SQLSTATE value in a local variable within the error handling procedure. Each subsequent SQL statement resets SQLSTATE, and you might lose a value you need for debugging.

Enabling and Disabling the WHENEVER Declaration

If you use `WHENEVER SQLERROR GOTO some_label;` to avoid infinite loops, enable the WHENEVER declaration at the beginning of the program and disable WHENEVER for the part of the program that handles the condition named in the declaration.

Example

This example enables and disables the WHENEVER directive:

```
C EXEC SQL WHENEVER SQLERROR GOTO end_prog; /* enables action */
...
end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;      /* disables action */
...
```

Saving and Restoring SQLSTATE or SQLCODE

If you use WHENEVER SQLERROR CALL sql_error and the sql_error function contains SQL statements, you can save and restore the value of SQLSTATE or SQLCODE before returning from the sql_error function.

Example

This example uses the first FETCH statement returning an SQLCODE value that is not equal to 0:

```
C EXEC SQL WHENEVER SQLERROR CALL sql_error;
...
EXEC SQL OPEN get_by_partnum;
...
EXEC SQL FETCH get_by_partnum
      INTO :hv_partnum, :hv_partdesc, :hv_price, :hv_qty_available;
...
while (SQLCODE == 0) {
    if ( hv_qty_available < 1000 )
        EXEC SQL UPDATE parts
              SET qty_available = qty_available + 100
              WHERE CURRENT OF get_by_partnum;
    ...
    EXEC SQL FETCH get_by_partnum
          INTO :hv_partnum, :hv_partdesc, :hv_price, :hv_qty_available;
}
...
void sql_error() {
    long saved_sqlcode = SQLCODE;
    EXEC SQL GET DIAGNOSTICS
          :hv_num = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
              :hv_sqlstate = RETURNED_SQLSTATE,
              :hv_msgtxt   = MESSAGE_TEXT;
    }
    ...
    SQLCODE = saved_sqlcode;
} /* end sql_error */
```

After the first FETCH statement returns an error, control moves to sql_error. Because the function sql_error contains SQL statements, the SQLCODE value in the function overwrites the global SQLCODE value. Before returning from the function, the global value is restored so that, when control is returned to the statement following

FETCH and SQLCODE is checked, the value is equal to the original value returned from FETCH.

Example

This example uses WHENEVER SQLERROR PERFORM to save and restore SQLSTATE:

COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 sqlstate          pic x(5).
    01 saved-sqlstate    pic x(5).
    01 hv-num            pic s9(9) comp.
    ...
EXEC SQL END DECLARE SECTION END-EXEC.
    ...
    EXEC SQL WHENEVER SQLERROR PERFORM sqlerrors END-EXEC.
    ...
sqlerrors.
    MOVE sqlstate TO saved-sqlstate.
    EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
    EXEC SQL GET DIAGNOSTICS
        :hv-num      = NUMBER
    END-EXEC.
    PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
        MOVE SPACES TO hv-msgtxt
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv-sqlstate = RETURNED_SQLSTATE,
            :hv-msgtxt   = MESSAGE_TEXT
        END-EXEC.
        DISPLAY "SQLSTATE: " hv-sqlstate
        DISPLAY "Message : " hv-msgtxt
        END-PERFORM.
    MOVE saved-sqlstate TO sqlstate.
    ...
END PROGRAM Program-exF72.
```

For information on using the GET DIAGNOSTICS Statement, see [Accessing and Using the Diagnostics Area](#) on page 13-17.

Declaring SQLSTATE or SQLCODE in an Error Routine

As described in [Saving and Restoring SQLSTATE or SQLCODE](#) on page 13-15, you can save and restore SQLSTATE or SQLCODE within the SQL error function. Alternately, in a C program, you can declare SQLSTATE in the function (in addition to declaring SQLSTATE in the main routine).

Example

This example program has two SQL Declare Sections, both of which contain an SQLSTATE declaration:

C

```
void sql_error(void);
...
int main ()
{
```

```

...
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
...
EXEC SQL END DECLARE SECTION;
...
EXEC SQL WHENEVER SQLERROR CALL sql_error;
...
EXEC SQL OPEN get_by_partnum;
EXEC SQL FETCH get_by_partnum
INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;
...
while (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    if ( hv_qty_available < 1000 )
        EXEC SQL UPDATE parts
            SET qty_available = qty_available + 100
            WHERE CURRENT OF get_by_partnum;
    ...
    EXEC SQL FETCH get_by_partnum
        INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;
}
...
return 0;
} /* end main */
...
void sql_error() {
    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    long hv_num;
    unsigned short i;
    char hv_sqlstate[6];
    VARCHAR hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL GET DIAGNOSTICS
        :hv_num = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_msgtxt = MESSAGE_TEXT;
        ...
    }
} /* end sql_error */

```

Accessing and Using the Diagnostics Area

NonStop SQL/MX stores completion and exception information in the diagnostics area. At the beginning of the execution of an SQL statement, the diagnostics area is emptied. When the statement executes, NonStop SQL/MX places information on completion or exception conditions in this area.

A transaction has a diagnostics area limit, which is a positive integer that specifies the maximum number of conditions that can be placed in the diagnostics area during

execution of an SQL statement within the transaction. Use the SET TRANSACTION statement to set the size of the diagnostics area.

To access the information in the diagnostics area, use the GET DIAGNOSTICS statement. The diagnostics area consists of:

- Statement information: Header area consisting of information on the SQL statement as a whole.
- Condition information: Detail area about each error, warning, or completion code that appeared during the execution of an SQL statement.

NonStop SQL/MX automatically allocates the diagnostics area in a program. You are not required to explicitly allocate it yourself.

For a description of the statement and condition items and the syntax, see the GET DIAGNOSTICS statement in the *SQL/MX Reference Manual*.

Using the GET DIAGNOSTICS Statement

Use this general syntax:

```
GET DIAGNOSTICS {statement-info | condition-info}
```

The *statement-info* is defined as:

```
target = stmt-item-name [, target = stmt-item-name]...
```

The *condition-info* is defined as:

```
EXCEPTION condition-number  
  target = condtn-item-name [, target = condtn-item-name]...
```

The *target* is a host variable that receives the requested diagnostics information. *target* must have the same data type as the *stmt-item-name* or *condtn-item-name* you are requesting. The *condition-number* specifies the number of an exception condition. It can be a literal or host variable with exact numeric data type.

Getting Statement and Condition Items

The next example uses the GET DIAGNOSTICS statement to display condition information after an INSERT statement. The first GET DIAGNOSTICS obtains the number of condition items. The second GET DIAGNOSTICS loops through the individual condition items and prints information for each condition.

See Statement Items-GET DIAGNOSTICS and Condition Items-GET DIAGNOSTICS in the *SQL/MX Reference Manual*.

You can retrieve the message text of the exception condition for SQLSTATE and SQLCODE. To provide a log of exception conditions, you can write the SQLSTATE and SQLCODE values, along with the message text, to a file for future reference.

Examples

```

C  /* Set new_jobcode and new_jobdesc host variables. */
EXEC SQL INSERT INTO sales.job (jobcode,jobdesc)
      VALUES (:new_jobcode,:new_jobdesc);

...
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
    { printf ("\nValues were inserted!");
      EXEC SQL COMMIT WORK; }
else get_diagnostics();

...
void get_diagnostics(void) {
EXEC SQL GET DIAGNOSTICS :num = NUMBER;

...
for (i = 1; i <= num; i++) {
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv_tabname   = TABLE_NAME,
      :hv_colname   = COLUMN_NAME,
      :hv_sqlstate  = RETURNED_SQLSTATE,
      :hv_sqlcode   = SQLCODE,
      :hv_msgtxt    = MESSAGE_TEXT;

    ...
    printf("Table    : %s\n", hv_tabname);
    printf("Column   : %s\n", hv_colname);
    printf("SQLSTATE: %s\n", hv_sqlstate);
    printf("SQLCODE  : %d\n", hv_sqlcode);
    printf("Message  : %s\n", hv_msgtxt);
    ... /* Process the SQL error. */
}

...
} /* end get_diagnostics */

```

```

COBOL * Set new-jobcode and new-jobdesc host variables.

...
EXEC SQL INSERT INTO sales.job (jobcode,jobdesc)
      VALUES (:new-jobcode,:new-jobdesc)
END-EXEC.

...
IF SQLSTATE = SQLSTATE_OK
    DISPLAY "Values were inserted!"
    EXEC SQL COMMIT WORK END-EXEC.
ELSE PERFORM 1000-GET-DIAGNOSTICS.

...
STOP RUN.
1000-GET-DIAGNOSTICS.
EXEC SQL GET DIAGNOSTICS
      :num = NUMBER
...
END-EXEC.
PERFORM VARYING i FROM 1 BY 1 UNTIL i > num
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv-tabname   = TABLE_NAME,
      :hv-colname   = COLUMN_NAME,
      :hv-sqlstate  = RETURNED_SQLSTATE,
      :hv-sqlcode   = SQLCODE,

```

```

        :hv-msgtext    = MESSAGE_TEXT
    END-EXEC.
    ...
    DISPLAY "Condition: " i
    DISPLAY "Table      : " hv-tabname
    DISPLAY "Column     : " hv-colname
    DISPLAY "SQLSTATE   : " hv-sqlstate
    DISPLAY "SQLCODE    : " hv-sqlcode
    DISPLAY "MESSAGE    : " hv-msgtext
    END-PERFORM.
* Process the SQL error
    ...
1000-GET-DIAGNOSTICS-END.

```

Special SQL/MX Error Conditions

Lost Open Error (8574)

When an embedded SQL program accesses a table or view by using a DML statement or an SQL cursor, NonStop SQL/MX opens the table or view and holds it open until the program stops executing or until the DML statement, if dynamically prepared, is deallocated. If the DML statement or cursor, or the transaction containing the statement or cursor, allows concurrent access to the table or view, the program could lose its open on the table or view to a DDL or SQL utility operation. The DDL or utility operation invalidates the open held by the program to change the structure of the table or view and gains exclusive access to the table or view. A program could also lose its open on a table or view when a network or hardware interruption occurs.

Occurrences of the Lost Open Error

If a DML statement partially modifies a database object (that is, table, view, and so on) before the open is invalidated, the SQL/MX executor rolls back the changes made by the statement and returns the Lost Open Error (8574) to the program. For example, consider an INSERT statement on a table that has an index. The INSERT statement always modifies the table first before updating the index. If the index is destroyed, the Lost Open Error occurs.

If a cursor returns one or more rows to the program before the open is invalidated, the SQL/MX executor returns the Lost Open Error (8574) to the program. If a cursor or DML statement does not return any rows to the program before the open is invalidated, the SQL/MX executor retries the cursor or DML statement and then waits for the lock to be released on the table or view. If the lock is not released before the timeout is reached, the SQL/MX executor returns the Lost Open Error (8574) to the program.

If the lock is released before the timeout is reached, the SQL/MX executor reopens the table or view. If reopening the table or view results in a timestamp mismatch, the SQL/MX executor performs a similarity check of the table or view. If the similarity check fails (or is disabled), the SQL/MX executor tries to automatically recompile the statement. If the SQL/MX executor cannot recompile the statement, it returns the Lost

Open Error (8574), as well as other recompilation errors, to the program. For more information on similarity checks and automatic recompilation, see [Section 8, Name Resolution, Similarity Checks, and Automatic Recompilation](#).

Recovering From the Lost Open Error

If DML or cursor operations in a program enable concurrent access to tables or views, or if you anticipate network or hardware interruptions, add code to the program to catch and handle the Lost Open Error (8574). The way that you handle occurrences of the Lost Open Error (8574) depends on what you are trying to accomplish with the DML statement or cursor. In most cases, when the Lost Open Error (8574) occurs, the program should retry the DML statement or close and reopen the cursor before executing a subsequent FETCH statement.

This example provides general error recovery code:

```
void sql_error(void)
{
    if (SQLCODE == -8574)
    {
        printf("Recovering cursor from error %ld\n", SQLCODE);
        EXEC SQL Open C1;
    }
}
```


14 Transaction Management

A transaction, which is a set of database changes that must be completed as a group, is the basic recoverable unit in case of a failure or transaction interruption.

The typical order of events is:

1. Transaction is started.
2. Database changes are made.
3. Transaction is committed if the database changes are made successfully.

If the transaction cannot successfully make the changes or if you do not want to complete the transaction, you can abort the transaction and roll the database back to its original state.

This section describes:

- [Transaction Control Statements](#) on page 14-1
- [Steps for Ensuring Data Consistency](#) on page 14-1

For more information on transaction management, see the *SQL/MX Reference Manual*.

Transaction Control Statements

Control the transactions in an C/C++/COBOL program by specifying these SQL/MX statements in an embedded SQL source file:

- BEGIN WORK statement
- COMMIT WORK statement
- ROLLBACK WORK statement
- SET TRANSACTION statement

For the syntax of these statements, see the *SQL/MX Reference Manual*. To use these transaction control statements in your C/C++/COBOL program, see [Steps for Ensuring Data Consistency](#) on page 14-1

Steps for Ensuring Data Consistency

[Figure 14-1](#) shows the steps presented within the complete C program. These steps are executed in the sample program [Example A-2](#) on page A-4.

C

Figure 14-1. Coding Transaction Control Statements in a C Program

```
1  ...
   char SQLSTATE_OK[6] = "00000";
   ...
   EXEC SQL BEGIN DECLARE SECTION;
   char SQLSTATE[6];
   ...
   EXEC SQL END DECLARE SECTION;
   ...
```

C**Figure 14-1. Coding Transaction Control Statements in a C Program**

```

2  /* Set attributes for the transaction. */
   EXEC SQL SET TRANSACTION
       READ WRITE,
       ISOLATION LEVEL SERIALIZABLE,
       DIAGNOSTICS SIZE 10;
   ...

3  EXEC SQL BEGIN WORK;           /* Begin the transaction. */
   ...

4  EXEC SQL UPDATE ... ;         /* Process database changes. */
   ...

5  if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) { /* Test if OK. */
   ...
   ...
6  EXEC SQL COMMIT WORK;         /* Commit database changes. */
   }
   else {
   ...
7  EXEC SQL ROLLBACK WORK; /* Rollback database changes. */
   }
   ...

```

[Figure 14-2](#) shows the steps presented within the complete COBOL program. These steps are executed in the sample program [Example C-2](#) on page C-4.

Figure 14-2. Coding Transaction Control Statements in a COBOL Program**COBOL**

```

1  ...
   01 SQLSTATE-OK   PIC X(5) VALUE "00000".
   ...
   EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 SQLSTATE      PIC X(5).
   ...
   EXEC SQL END DECLARE SECTION END-EXEC.
   ...

2  * Set attributes for the transaction.
   EXEC SQL SET TRANSACTION
       READ WRITE,
       ISOLATION LEVEL SERIALIZABLE,
       DIAGNOSTICS SIZE 10
   END-EXEC.
   ...

3  * Begin the transaction.
   EXEC SQL BEGIN WORK END-EXEC.
   ...

4  * Process database changes.
   EXEC SQL UPDATE ... END-EXEC.
   ...

```

Figure 14-2. Coding Transaction Control Statements in a COBOL Program**COBOL**

```

5  * Test if database change is ok.
   IF SQLSTATE = SQLSTATE-OK
   ...
6  * Commit database change.
   EXEC SQL COMMIT WORK END-EXEC.

   ELSE
   ...
7  * Roll back database change.
   EXEC SQL ROLLBACK WORK END-EXEC.
   END-IF.
   ...

```

For more information, see:

1. [Declaring Required Variables](#) on page 14-3
2. [Setting Attributes for Transactions](#) on page 14-3
3. [Starting a Transaction](#) on page 14-6
4. [Processing Database Changes](#) on page 14-7
5. [Testing for Errors](#) on page 14-7
6. [Committing Database Changes if No Errors Occur](#) on page 14-8
7. [Undoing Database Changes if an Error Occurs](#) on page 14-8

Declaring Required Variables

Declare SQLSTATE and host variables you need within your program.

Examples

C

```

char SQLSTATE_OK[6] = "00000";
EXEC SQL BEGIN DECLARE SECTION;
char SQLSTATE[6];
...
EXEC SQL END DECLARE SECTION;

```

COBOL

```

01 SQLSTATE-OK PIC X(5) VALUE "00000".
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQLSTATE PIC X(5).
...
EXEC SQL END DECLARE SECTION END-EXEC.

```

Setting Attributes for Transactions

Use a SET TRANSACTION statement to set some of the attributes for subsequent transactions. When coding a SET TRANSACTION statement in an embedded SQL program, consider that special considerations exist for setting the isolation level. See [Isolation Level Setting](#) on page 14-5.

Example

The attributes include access mode and isolation level, which affect the degree of concurrent data access:

C

```
...
EXEC SQL SET TRANSACTION
    READ ONLY,
    ISOLATION LEVEL READ UNCOMMITTED,
    DIAGNOSTICS SIZE 10;
...
```

The last attribute shown in this example, the diagnostics size, is an estimate of the number of exception conditions you might expect as a result of executing an SQL statement within the transaction. Specifically, the diagnostics size is the maximum number of condition items for a statement item.

To set attributes for transactions in an C/C++/COBOL program, see:

- [Autocommit Setting](#) on page 14-4
- [Isolation Level Setting](#) on page 14-5
- [Default Transaction Attributes](#) on page 14-6

Autocommit Setting

The autocommit setting specifies whether an implicit transaction commits automatically at the end of statement execution or rolls back automatically if an error occurs. When autocommit is enabled (that is, turned on), each implicit transaction commits automatically at the end of statement execution.

The BEGIN WORK statement turns off the autocommit setting inside an explicit, user-defined transaction. Even if you turn on the autocommit setting before an explicit transaction starts, you must code a COMMIT WORK statement to commit the transaction. By default, autocommit is off in C/C++/COBOL programs. To turn on autocommit, issue a SET TRANSACTION statement in your application. For more information on autocommit, see the *SQL/MX Reference Manual*.

Use SET TRANSACTION at the beginning of your C program to commit changes automatically at the end of each SQL statement:

```
...
EXEC SQL SET TRANSACTION AUTOCOMMIT ON;
...
```

Use a separate SET TRANSACTION statement to set AUTOCOMMIT ON. You cannot specify this option in combination with any other option in the SET TRANSACTION statement.

See the SET TRANSACTION statement in the *SQL/MX Reference Manual*.

Isolation Level Setting

The isolation level specifies the level of data consistency defined for the transaction and the degree of concurrency the transaction has with other transactions that use the same data. The isolation level of a transaction can be READ UNCOMMITTED, READ COMMITTED, or SERIALIZABLE (or REPEATABLE READ). For more information, see the *SQL/MX Reference Manual*.

In an embedded SQL program, avoid using the SET TRANSACTION ISOLATION LEVEL statement because:

- It always starts a compiler process.
- It could cause automatic recompilation.
- It could adversely affect UPDATE, INSERT, and DELETE statements at run time within its control flow scope.

The SET TRANSACTION ISOLATION LEVEL statement, regardless of whether it was statically compiled, always executes dynamically and applies its setting at run time. Therefore, this statement always starts a compiler process. To avoid performance costs, use a CONTROL QUERY DEFAULT ISOLATION_LEVEL statement instead. For more information on coding CONTROL statements, see [Using CONTROL Statements](#) on page 2-12.

The SET TRANSACTION ISOLATION LEVEL statement could cause automatic recompilation of DML statements in the next transaction. If a DML statement is statically compiled and does not specify an explicit access option (for example, READ COMMITTED, SERIALIZABLE, and so on), its access option at compile time is determined by the ISOLATION_LEVEL setting, if present, or by the system-defined isolation level, which is READ COMMITTED. If NonStop SQL/MX executes the DML statement after executing a SET TRANSACTION ISOLATION LEVEL statement with a different isolation level setting, the SQL/MX executor automatically recompiles the DML statement. To avoid automatic recompilation, specify explicit access options in individual DML statements. The access options in DML statements override the isolation level of any containing transactions. See [Precedence of Transaction Isolation Levels](#) on page 14-6.

If you set the isolation level to READ UNCOMMITTED, the access mode becomes READ ONLY by default. As a result, INSERT, UPDATE, and DELETE statements within the scope of a SET TRANSACTION or CONTROL QUERY DEFAULT statement fail to compile or execute. INSERT, UPDATE, and DELETE statements require the access mode to be READ WRITE. To avoid compilation or run-time errors, make sure that subsequent transactions do not contain INSERT, UPDATE, and DELETE statements if you specify READ UNCOMMITTED in a SET TRANSACTION or CONTROL QUERY DEFAULT statement. Instead of using those statements, consider specifying READ UNCOMMITTED as the access option in each SELECT statement.

Precedence of Transaction Isolation Levels

NonStop SQL/MX determines the transaction isolation level, based on these settings, in order of precedence, from highest to lowest:

1. If you specify an access option explicitly in a DML statement, the SQL/MX compiler compiles the statement with the access option. This access option overrides the isolation level of any containing transactions.
2. If there are no individual statement access options and you issue a SET TRANSACTION ISOLATION LEVEL statement, the SQL/MX compiler uses the setting determined by this SET TRANSACTION statement as the isolation level for the next transaction.
3. If you do not specify a SET TRANSACTION statement and you issue a CONTROL QUERY DEFAULT ISOLATION_LEVEL statement, the CONTROL QUERY DEFAULT statement determines the isolation level.
4. If you do not issue a CONTROL QUERY DEFAULT ISOLATION_LEVEL statement, NonStop SQL/MX uses the ISOLATION_LEVEL setting in the SYSTEM_DEFAULTS table if it exists.
5. If you do not specify isolation-level settings, NonStop SQL/MX uses the system-defined isolation level, which is READ COMMITTED.

Default Transaction Attributes

If you do not explicitly set the transaction attributes, the embedded SQL program uses these default attributes for the next transaction in a program:

- Autocommit: OFF by default in embedded SQL in C and COBOL programs. ON by default at the start of an MXCI session.
- Isolation Level: READ COMMITTED
- Access mode: READ WRITE
- Size of diagnostics area: Thirty conditions

Starting a Transaction

Use a BEGIN WORK statement to start a transaction explicitly:

```
EXEC SQL BEGIN WORK;
```

The transaction consists of the sequence of SQL statements that begins immediately after BEGIN WORK and ends with the next COMMIT or ROLLBACK statement.

If you do not use the BEGIN WORK statement, NonStop SQL/MX automatically starts a transaction for a statement, provided that an active transaction does not already exist and that the statement supports implicit transactions. For information on implicit (or system-defined) transactions, see the *SQL/MX Reference Manual*.

In a program, you might use a loop when updating or deleting rows in the result set of a cursor. In a looping UPDATE or DELETE (either searched or positioned), NonStop SQL/MX commits changes as they occur within the loop when autocommit is on. To ensure database consistency when autocommit is on, issue BEGIN WORK before the loop starts (or before declaring the cursor) and issue COMMIT WORK after all changes have been made within the loop. For more information, see the [Autocommit Setting](#) on page 14-4 and [Committing Database Changes if No Errors Occur](#) on page 14-8.

Note. By default, autocommit is off in C/C++/COBOL programs.

For the syntax of the BEGIN WORK statement, see the *SQL/MX Reference Manual*.

Processing Database Changes

A transaction typically consists of a sequence of SQL statements that change the database. For example, a transaction might include INSERT, DELETE, or UPDATE statements:

```
EXEC SQL UPDATE ... ;
```

Typically, the SQL statements within a single transaction are dependent on each other. For example, suppose that you want to change a job code in your database. You might insert the new job code in the JOB table, update the job code in the EMPLOYEE table, and finally delete the old job code in the JOB table. These operations are logically dependent on one another and, therefore, you should group them within one user-defined transaction.

You should avoid certain DML operations on the same set of rows as a cursor operation in the same transaction. The sensitivity of cursors is ASENSITIVE in NonStop SQL/MX, which means that a concurrent DML operation in the same transaction as a cursor might or might not affect the cursor results. For information on the DML operations to avoid, see [Cursor Sensitivity](#) on page 6-16.

If you include a CALL statement within a transaction, be aware that the stored procedure in Java (SPJ) invoked by the CALL statement inherits the transaction from the caller, and the SPJ method cannot contain transaction control statements. For more information, see the *SQL/MX Guide to Stored Procedures in Java*.

Testing for Errors

You can test SQLSTATE for a return value of 00000 (successful completion).

Examples

C `if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) ...`

COBOL `IF sqlstate = sqlstate-ok ...`

However, the diagnostics area provides more information than the SQLSTATE variable because condition information is stored for each exception condition that occurs during execution of an SQL statement.

Note. The results of executing an SQL statement overlay the results of the previous SQL statement in the diagnostics area. Therefore, test for exception conditions after the execution of each statement within your transaction.

Committing Database Changes if No Errors Occur

The COMMIT WORK statement permanently commits changes made to the database within the current transaction and ends the transaction. It frees resources held by the transaction, such as row or table locks.

```
EXEC SQL COMMIT WORK;
```

If the program changes the database, either issue a COMMIT WORK statement at the end of your transaction or turn on autocommit. If you quit a program without committing a transaction, database changes are automatically rolled back.

For the syntax of the COMMIT WORK statement, see the *SQL/MX Reference Manual*.

Undoing Database Changes if an Error Occurs

The ROLLBACK WORK statement undoes changes made to the database within the current transaction and ends the transaction. It frees resources held by the transaction, such as row or table locks.

```
EXEC SQL ROLLBACK WORK;
```

For the syntax of the ROLLBACK WORK statement and more information on transaction management, see the *SQL/MX Reference Manual*.

15 C/C++ Program Compilation

This section describes how to develop and execute a C/C++ program that contains embedded SQL statements. In addition, this section contains information on embedded module definitions and module definition files:

- [Compiling SQL/MX Applications and Modules](#) on page 15-2
- [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8
- [Running the C/C++ Compiler and Linker](#) on page 15-34
- [Running the SQL/MX Compiler](#) on page 15-36
- [c89 Utility: Using One Command for All Compilation Steps](#) on page 15-44
- [Examples of Building and Deploying Embedded SQL C/C++ Programs](#) on page 15-55
- [Building SQL/MX C/C++ Applications to Run in the Guardian Environment](#) on page 15-66
- [Running an SQL/MX Application](#) on page 15-72

For information on managing C/C++ programs and SQL/MX modules, see [Section 17, Program and Module Management](#).

Compiling SQL/MX Applications and Modules

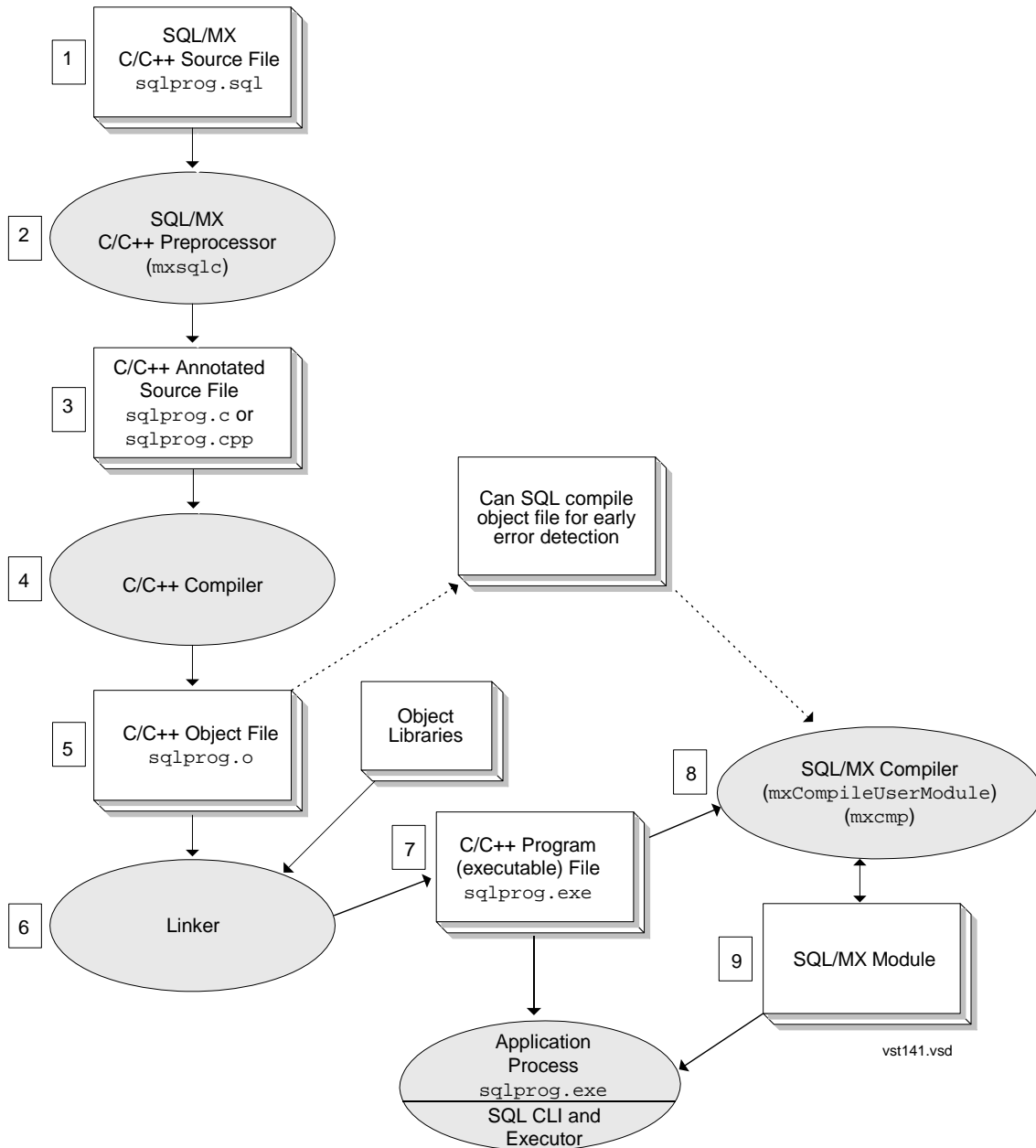
NonStop SQL/MX Release 2.x provides two methods of compiling embedded SQL C/C++ programs and creating modules. Both methods create an identical module file. The first method described, using embedded module definitions, is the default and preferred method.

The SQL/MX preprocessor reads a source file that contains C/C++ and embedded SQL statements and generates:

- | | |
|--|--|
| Method 1: Embedded module definitions | One file: a single, self-contained annotated source file that contains source statements with SQL statements converted to comments and embedded module definitions. You compile this file (<i>source-file.c</i> in embedded SQL/MX C programs or <i>source-file.cpp</i> in embedded SQL/MX C++ programs) with the C/C++ compiler (c89) and the SQL/MX compiler (mxCompileUserModule). This is the default and preferred method. |
| Method 2: Annotated source file and module definition file | Two files: an annotated source file and a module definition file (<i>source-file.m</i>) that contains SQL source statements. You compile the source file with the C/C++ compiler, and you compile the module definition file with the SQL/MX compiler (mxcmp). A module definition file is not created unless you use the <code>-x</code> or <code>-m</code> preprocessor options or set the <code>SQLMX_PREPROCESSOR_VERSION=800</code> environment variable to create a module definition file. For more information, see Influencing Module Management Behavior on page 17-9. |

Compiling Embedded SQL C/C++ Programs With Embedded Module Definitions

[Figure 15-1](#) on page 15-3 shows how a self-contained, single-file C/C++ program is compiled using embedded module definitions. The application's embedded SQL source file is called `sqlprog.sql`.

Figure 15-1. Compiling Embedded SQL C/C++ Programs With Embedded Module Definitions

Although this figure shows individual steps for clarity, you can use `c89` or the HP Enterprise Toolkit—NonStop Edition (ETK) to automate the process. For information on using `c89` in this way, see [c89 Utility: Using One Command for All Compilation Steps](#) on page 15-44. For more information on using ETK, see ETK online help.

These steps correspond to the steps in [Figure 15-1](#) on page 15-3.

1. Create the C or C++ source file that contains embedded SQL statements (`sqlprog.sql`).

2. Preprocess the application's embedded SQL source files by using the SQL/MX C/C++ preprocessor `mxsqlc`. See [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

```
mxsqlc sqlprog.sql -c sqlprog.c
```

In this step, set optional module specification strings and `moduleCatalog` and `moduleSchema` default settings by using the `-g` option. See [15-23](#) or [15-32](#).

Although you do not set `mxcmp` defaults here, if the input source file contains `mxcmp` default settings, such as `EXEC SQL DECLARE/SET/CONTROL QUERY DEFAULT` statements, they are preprocessed into corresponding module language statements in the output module definition of the annotated source file.

3. The preprocessor produces a modified (annotated) C source file (`sqlprog.c` in C or `sqlprog.cpp` in C++) that contains the C and SQL call-level interface (CLI) translations of embedded SQL statements and additional C/C++ source constructs that represent the module definition. The default behavior creates a single, self-contained application source file with embedded module definitions.
4. Compile the annotated C/C++ source file by using the `c89` compiler (HP NonStop Open System Services (OSS) environment) or ETK (Windows environment). To produce an object file:

```
c89 -c sqlprog.c -o sqlprog.o
```

Specify the `-c` option if you do not want `c89` to link the program. Otherwise, `c89` invokes `eld` or `nld` to create an executable file.

See [Running the C/C++ Compiler and Linker](#) on page 15-34.

5. The C/C++ compiler produces the object file, `sqlprog.o`. If you prefer early detection of SQL compilation errors, you can SQL compile the application's object file at this point. During program development, you might want to use the `mxCompileUserModule` utility against all the object files rather than against the executable file. When you SQL compile against the object files, NonStop SQL/MX does not recompile each module for object files that are linked into more than one executable file.
6. Link application object files with object libraries to create an executable file.

- For TNS/E native compilation, use the `eld` utility:

```
eld /usr/lib/ccplmain.o sqlprog.o \
-o sqlprog.exe -lzcredll -lzctrlldll -lzoskdll \
-lzi18ndll -lzicnvdl1 -lzclidll
```

- For TNS/R native compilation, use the `nld` utility:

```
nld /usr/lib/crtlmain.o sqlprog.o -o sqlprog.exe -elf \
-set systype oss -set highpin off -set highrequestor on \
-set inspect on -obey /usr/lib/libc.obey \
-set saveabend on \
-Bdynamic -lzcp1srl -lzctrlsrl -lzcre srl -lzcplosrl \
-lztlhg srl -lzt1hosrl -lzclisrl
```

7. The linker produces the application's executable file, `sqlprog.exe`.
8. SQL compile one, some, or all of the application's embedded module definitions in the executable file by using `mxCompileUserModule`. See [Running the SQL/MX Compiler](#) on page 15-36 and [Compiling Embedded Module Definitions](#) on page 15-37.

```
mxCompileUserModule sqlprog.exe
```

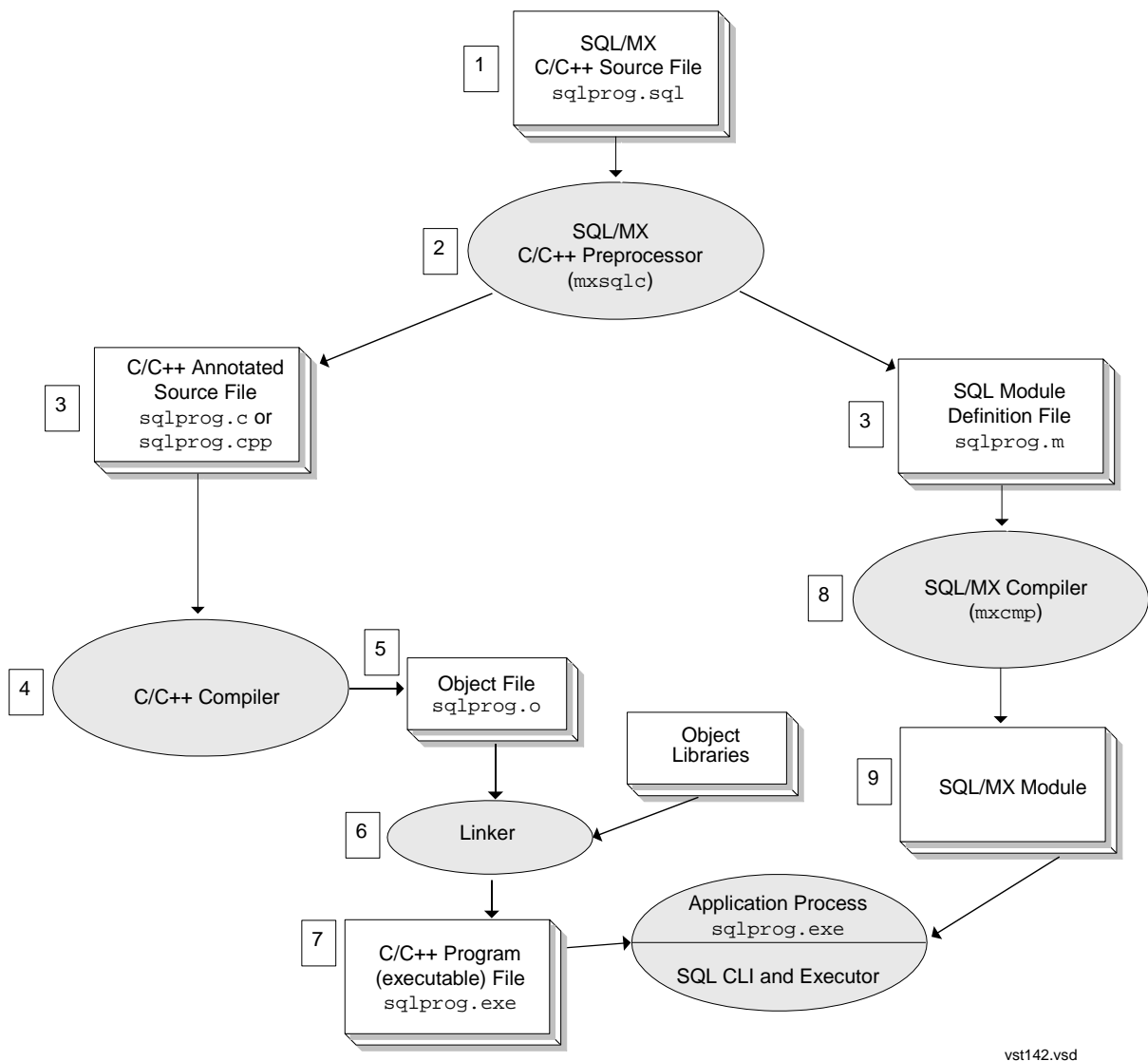
9. The SQL/MX compiler produces the SQL/MX module. The module is stored in the local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.

Run the C/C++ executable program.

Compiling Embedded SQL C/C++ Programs With Module Definition Files

[Figure 15-2](#) shows how a C/C++ program with separate module definition files is compiled. The application's embedded SQL source file is called `sqlprog.sql`.

Figure 15-2. Compiling Embedded SQL C/C++ Programs With Module Definition Files



Although this figure shows individual steps for clarity, you can use the `c89` utility or ETK to automate the process. For more information on using `c89` in this way, see [c89 Utility: Using One Command for All Compilation Steps](#) on page 15-44. For more information on ETK, see ETK online help.

These steps correspond to the steps in [Figure 15-2](#) on page 15-6.

1. Create the C or C++ source files that contain embedded SQL statements (sqlprog.sql).
2. Preprocess the application's embedded SQL source files by using the SQL/MX C/C++ preprocessor `mxsqlc`. See [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

```
mxsqlc sqlprog.sql -c sqlprog.c -m sqlprog.m
```

In this step, set optional module specification strings and `moduleCatalog` and `moduleSchema` default settings by using the `-g` option. See [15-23](#) or [15-32](#). Although you do not set `mxcmp` defaults here, if the input source file contains `mxcmp` default settings, such as `EXEC SQL DECLARE/SET/CONTROL QUERY DEFAULT` statements, they are preprocessed into corresponding module language statements in the output module definition of the module definition file. The preprocessor options (`-x` or `-m`) and the `SQLMX_PREPROCESSOR_VERSION=800` environment variable indicate to the preprocessor that you are compiling your program with module definition files. For more information on setting the preprocessor options, see [Module Management Behavior](#) on page 17-8.

3. The preprocessor produces two files: a modified (annotated) C source file (sqlprog.c in C or sqlprog.cpp in C++) that contains the C and SQL CLI translations of embedded SQL statements and the module definition file (sqlprog.m).
4. Compile the annotated C/C++ source file by using the `c89` compiler (OSS environment) or `ETK` (Windows environment). To produce an object file:

```
c89 -c sqlprog.c -o sqlprog.o
```

Specify the `-c` option if you do not want `c89` to link the program. Otherwise, `c89` invokes `eld` or `nld` to create an executable file.

See [Running the C/C++ Compiler and Linker](#) on page 15-34.

5. The C/C++ compiler produces the ELF object file, sqlprog.o.
6. Link application object files with object libraries to create an executable file.

- For TNS/E native compilation, use the `eld` utility:

```
eld /usr/lib/ccplmain.o sqlprog.o \
  -o sqlprog.exe -lzcredll -lzcrtdll -lzoskdll \
  -lzi18ndll -lzicnvdl1 -lzclidll
```

- For TNS/R native compilation, use the `nld` utility:

```
nld /usr/lib/crtlmain.o sqlprog.o -o sqlprog.exe -elf \
  -set systype oss -set highpin off -set highrequestor on \
  -set inspect on -obey /usr/lib/libc.obey \
  -set saveabend on \
  -Bdynamic -lzcp1srl -lzcr1srl -lzcrsrl -lzcp1osrl \
  -lzt1hsrl -lzt1hosrl -lzt1isrl
```

7. The linker produces the application's executable file, sqlprog.exe.

8. SQL compile the application's module definition file by using the SQL/MX compiler (mxcmp). See [Running the SQL/MX Compiler](#) on page 15-36 and [Compiling a Module Definition File](#) on page 15-42.

```
mxcmp sqlprog.m
```

9. The SQL/MX compiler compiles the SQL source statements from the module definition file into a module file, generates a SQL object code for each statement, determines an optimized execution plan for each SQL statement against the database, and then stores the code and plan in the SQL object program. The module is stored in the local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global /usr/tandem/sqlmx/USERMODULES directory.

Run the C/C++ executable program.

Creating Modules: From Development to Production

While HP recommends that you use embedded module definitions to create SQL modules, you might find it easier for debugging purposes to use module definition files during early stages of development and then switch to embedded module definitions upon deployment of your production system. Consider this:

- With embedded module definitions, you must successfully compile the output from the preprocessor before you can SQL compile the embedded module definition. You must diagnose host language errors in the source program before you can diagnose SQL errors in the source program.
- With module definition files, you compile the source file and the module definition file at the same time. This method provides the opportunity to diagnose both host language errors and SQL errors in the source file concurrently.

Embedded module definitions provide greater efficiency in the deployment of an application to a production environment.

Running the SQL/MX C/C++ Preprocessor

The SQL/MX C/C++ preprocessor is available for these environments:

- OSS
- Microsoft Windows
- Enterprise Plugins for Eclipse (EPE)
- Enterprise ToolKit—NonStop Edition (ETK)

Note. ETK is a GUI-based extension package to the Visual Studio .NET product. Use ETK to edit, compile, build, and deploy applications written in a variety of programming languages with embedded SQL/MX. For more information, see ETK online help.

The preprocessor for the OSS environment is installed when you install NonStop SQL/MX on your system. You must install the Windows-hosted preprocessor on your

Windows workstation. For information, see the *SQL/MX Release 3.2 Installation and Upgrade Guide*.

The syntax for using the preprocessor in each environment appears under [Syntax for the OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-20 and [Syntax for the Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-28.

Preprocessor Functions

The preprocessor processes C/C++ and SQL source statements.

C/C++ Source Statements

The preprocessor writes each C/C++ source statement to the C/C++ annotated source file. The preprocessor parses the source file only to the extent necessary to detect scoping levels, host variable declarations, host variable expressions, and embedded SQL statements.

C Preprocessing Directives

The preprocessor ignores the C directives except the `#include`, `#define`, `#line`, `#pragma`, or conditional compilation directives.

The pragma `#pragma SQL CHAR_AS_ARRAY` – SQL/MX Release 3.1, supports the `CHAR_AS_ARRAY` pragma in the embedded SQL file. The length of all character descriptors will be the same as the descriptors defined in the application, unlike the default length, where the length is one byte less than that defined.

The pragma can be placed anywhere in the file. However, the pragma will be effective from one definition until the next or until the end of file.

The following example uses the pragma `SQL CHAR_AS_ARRAY`:

```
#pragma SQL CHAR_AS_ARRAY
EXEC SQL BEGIN DECLARE SECTION;
char a[20];
EXEC SQL END DECLARE SECTION;
---
strcpy(a, "abc");
EXEC SQL INSERT INTO t1 (val) values(:a);
```

The following is the content of the generated module file:

```
ALLOCATE STATIC INPUT DESCRIPTOR
SQLMX_DEFAULT_STATEMENT_1_0_IVAR FOR STATEMENT
SQLMX_DEFAULT_STATEMENT_1 (CHARACTER(20) NOT NULL);

----- STATEMENT INDEX 0 -----

PROCEDURE SQLMX_DEFAULT_STATEMENT_1 ("a" CHARACTER(20))  INSERT
INTO t1 (val) values(:"a");
```

The pragma, #pragma SQL CHAR_AS_STRING – SQL/MX Release 3.1, supports the CHAR_AS_STRING pragma in the embedded SQL file. The length of all character descriptors will be one byte less than that defined in the application. This is the default behavior.

The pragma can be placed anywhere in the file. However, the pragma will be effective from one definition until the next, or until the end of file.

The following example uses the pragma SQL CHAR_AS_STRING:

```
#pragma SQL CHAR_AS_STRING
EXEC SQL BEGIN DECLARE SECTION;

char a[20];

EXEC SQL END DECLARE SECTION;

---

strcpy(a, "ramu");

EXEC SQL INSERT INTO t1 (val) values(:a);
```

The following is the content of the generated module file:

```
ALLOCATE STATIC INPUT DESCRIPTOR
SQLMX_DEFAULT_STATEMENT_1_0_IVAR FOR STATEMENT
SQLMX_DEFAULT_STATEMENT_1 (CHARACTER(19) NOT NULL);

----- STATEMENT INDEX 0 -----

PROCEDURE SQLMX_DEFAULT_STATEMENT_1 ("a" CHARACTER(19))  INSERT
INTO t1 (val) values(:"a");
```

Note. INVOKE includes its own option to support CHAR_AS_ARRAY and CHAR_AS_STRING. When the pragma is defined at the file level and the INVOKE option is not defined, the pragma definition is considered. However, when the pragma and the INVOKE option are defined, the INVOKE option will override the pragma definition.

C #include directive

The preprocessor expands first-level #include files. The original #include line is commented in the output source file. The commented line is followed by the #include file contents.

If you specify the `-I` option, the preprocessor expands the nested `#include` files. The `-I` option supports a maximum nesting limit of 200 levels. While processing nested `#include` files, circular inclusion of `include` files is detected, and the preprocessor issues warning 13089, and comments the `#include` line in the output source file.

By default, the preprocessor processes only the `#include` files on OSS, that have a `.mxh` extension. Using the `-h` command-line option, the preprocessor processes the `#include` files with any extension or no extension. The preprocessor also processes the `#include` files specified within the `pragma MXH` and `NOMXH` directives. It supports Guardian `DEFINEs` for the `#include` directive. If the `-O` option is specified, the OSS-hosted SQL/MX preprocessor resolves the Guardian class `MAP DEFINE` with the actual filename and processes it. The Windows-hosted SQL/MX preprocessor does not support Guardian `DEFINEs`.

Note. The `DEFINEs` are resolved only if the preprocessor option `-O` is specified.

The preprocessor ignores:

- Nested `#include` directives, if the `-I` option is not specified
- System `#include` directives (for example, `<time.h>`)
- The `NOLIST` option if it is part of the `#include` file command

Examples:

- The contents of the `mine3.mhx` file are included in the output source file:

```
#include "mine3.mhx"
```
- Only `sect1`, `sect2`, and `sect6` are included, and `NOLIST` is ignored:

```
#include "mine4.mhx (sect1,sect2,sect6)" NOLIST
```
- The contents of the `mine.h` and `mine2` files are included if `-h` is specified, and `NOLIST` is ignored:

```
#include "mine.h"
#include "../includes/mine2" NOLIST
```
- The contents of the file mapped by `DEFINE =cdef1` are included, if `-h` and `-O` are specified:

```
#include "=cdef1"
```

- If the `-I` option is specified, the contents of `a1.mhx`, `a2.mhx`, and `a3.mhx` are included in the output source file:

```
a1.sql
|
+-->a1.mhx
    |
    +-->a2.mhx
        |
        +-->a3.mhx
```

The following scenarios explain the behavior of the nested `#include` files:

In case 1, while processing nested `#include` files, the `#include` file `a1.mhx` which is included circularly is not processed, and a warning 13089 is returned by the preprocessor. The circular `#include` line is commented in the output source file.

Case 1:

```
a1.sql
|
+-->a1.mhx
    |
    +--->a2.mhx
        |
        +-->a1.mhx
```

In case 2, the `#include` file `b1.mhx` is not processed by the preprocessor.

Case 2:

```
b1.sql
|
+-->b1.mhx
    |
    +-->b1.mhx
```

As with any `#include` file inclusion, you must ensure that implementation of conditional compilation does not result in repeated file inclusion.

When a C `#pragma` directive is contained in an included file, it is processed by the preprocessor.

If the file `incl.mhx` contains:

```
#pragma section sect1
EXEC SQL BEGIN DECLARE SECTION;
  int a1;
EXEC SQL END DECLARE SECTION;

#pragma section sect2
EXEC SQL BEGIN DECLARE SECTION;
  int a2;
EXEC SQL END DECLARE SECTION;

#pragma section sect3
EXEC SQL BEGIN DECLARE SECTION;
  int a3;
EXEC SQL END DECLARE SECTION;
```

this construct:

```
#include "incl.mhx (sect1, sect3)" NOLIST
```

is expanded to:

```
/* #include "incl.h (sect1, sect3)" NOLIST */
EXEC SQL BEGIN DECLARE SECTION;
  int a1;
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN DECLARE SECTION;
  int a3;
EXEC SQL END DECLARE SECTION;
```

This construct:

```
#include "incl.mhx" NOLIST
```

is expanded to:

```
/* #include "incl.h" NOLIST */
#pragma section sect1
EXEC SQL BEGIN DECLARE SECTION;
int a1;
EXEC SQL END DECLARE SECTION;

#pragma section sect2
EXEC SQL BEGIN DECLARE SECTION;
int a2;
EXEC SQL END DECLARE SECTION;

#pragma section sect3
EXEC SQL BEGIN DECLARE SECTION;
int a3;
EXEC SQL END DECLARE SECTION;
```

C #Pragma MXH and #Pragma NOMXH Directive

The preprocessor processes all the user header files (for example, `#include "file1.h"`), that are within the `pragma` directives `MXH` and `NOMXH`, regardless of the header file extension.

Example:

The contents of the `mine.h` and `mine3` files are not included in the output source file. The content of the `../includes/mine2` file is processed and written to the output source file:

```
#include "mine.h"
#pragma MXH
#include "../includes/mine2"
#pragma NOMXH
#include "mine3"
```

C #define Directive

The preprocessor scans all `#define` directives and stores them in a table for evaluation when they are encountered. The preprocessor evaluates the stored defines for all legal combinations of conditional compilation.

A `#define` specified on the preprocessor command line must also be specified on the C/C++ command line. The preprocessor interprets and uses `#define` information but does not remove it from the generated code. The C/C++ compiler must get the same directive to interpret the code the same way. If you use the `c89` utility, this is not a concern.

The preprocessor checks each nonkeyword that begins a line to determine if it is in the define table. If it is, it is expanded. However, you must ensure that define-engendered substitutions result in valid code.

This `#define` directive:

```
#define SQL_Control_Table(defname)
    EXEC SQL CONTROL TABLE defname TABLELOCK 'OFF';

SQL_Control_Table(fldrenty);
SQL_Control_Table(postact);
SQL_Control_Table(permdeny);
```

is expanded to:

```
EXEC SQL CONTROL TABLE fldrenty TABLELOCK 'OFF';
EXEC SQL CONTROL TABLE postact TABLELOCK 'OFF';
EXEC SQL CONTROL TABLE permdeny TABLELOCK 'OFF';
```

The preprocessor also expands `#define` directives that occur within host variable parameters.

This `#define` directive:

```
#define MAX 255

EXEC SQL BEGIN DECLARE SECTION;
char mystr [MAX-1];
EXEC SQL END DECLARE SECTION;
```

is expanded to:

```
#define MAX 255

EXEC SQL BEGIN DECLARE SECTION;
char mystr [/*MAX-1*/ 254];
EXEC SQL END DECLARE SECTION;
```

C #line Directive

The preprocessor generates `#line` directives in the C/C++ annotated source file so that the user, during debugging, is directed to the input source line number and file name instead of the preprocessor-generated code that implements the embedded SQL statement. The preprocessor uses the source line number and input file name when reporting error and warning messages.

If the preprocessor encounters a `#line` directive, it updates the current source line number and input file name (if specified) from the directive.

C/C++ Comments

The preprocessor ignores C and C++ comments unless the comment specifies a name for an SQL statement. You can use a comment to name an SQL statement explicitly. To do so, precede the statement with a C comment using the format:

```
/* SQL statement_name = name [ comment-text ] */
EXEC SQL sql_statement ... ;
```

The *name* is an SQL identifier you are assigning as the name of *sql_statement*, and *comment-text* is an optional comment that does not affect the assignment of the name. The C/C++ comment must use only one line and must immediately precede the SQL statement.

For example, this comment names the SQL statement (INSERT) and provides comment text ("insert ten rows"):

```
/* SQL statement_name= INSERT insert ten rows */
EXEC SQL INSERT INTO ... ;
```

If you do not specify a name for an SQL statement, the preprocessor assigns the statement a name of the form `SQLMX_DEFAULT_STATEMENT_n`, where *n* is an integer incremented by the preprocessor.

Host Variable Declarations

The preprocessor checks each host variable declaration (that is, a variable declared between `BEGIN DECLARE SECTION` and `END DECLARE SECTION`) to ensure that the variable uses a valid data type. For valid host-variable data types, see [Table 3-1](#) on page 3-9 and [Table 3-4](#) on page 3-12.

The preprocessor parses `INVOKE` as a valid embedded SQL statement within a host variable declaration section. The preprocessor returns an error for embedded SQL statements that are not valid within a host-variable declaration section.

`SQLSTATE` must be declared within a Declare Section. See [Declaring SQLSTATE](#) on page 13-2.

Floating-Point Format

In SQL/MX Release 2.x, the preprocessor operates in ANSI IEEE floating-point format. In previous releases, the preprocessor used Tandem floating-point format. If you have applications that use floating-point data types and host variables, see [Assigning Floating-Point Data Types](#) on page 3-33.

Executable SQL Statements

The preprocessor performs these functions:

- Scans the statement for host variables (indicated by a colon) and ensures that each host variable is declared within the current scope of the program.
- Converts the SQL statement to a C comment in the C/C++ annotated source file.
- Writes the appropriate CLI procedure call or calls for the SQL statement immediately after the commented statement in the C/C++ annotated source file. At run time, the calls invoke the SQL/MX executor to execute the procedure for the SQL statement within the module.
- Writes the executable SQL statement to a separate module definition file if you use the `-x` or `-m` preprocessor option or set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable.

Use the preprocessor to embed SQL anywhere in the C/C++ source file. However, the preprocessor determines in which part of the source file the embedded SQL is located and issues warnings if an embedded SQL statement is not placed correctly. See [Placement of SQL Statements](#) on page 2-2.

At the end of processing the embedded SQL C/C++ source file, the preprocessor checks the status of static cursors:

- Cursors accessed and not opened return an error message.
- Cursors declared and not accessed return a warning message.

Preprocessor Output

C/C++ Annotated Source File for Embedded Module Definitions

The SQL/MX C/C++ preprocessor processes a C/C++ source file, such as *source-file.sql*, and generates one annotated source file (*source-file.c* in C or *source-file.cpp* in C++) as its output file. The annotated source file contains the embedded module definitions.

C/C++ Annotated Source File for Module Definition Files

If you use the `-x` or `-m` preprocessor option or if you set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable, the preprocessor processes a C/C++ source file, such as *source-file.sql*, and generates two files: the annotated source file (*source-file.c* in C or *source-file.cpp* in C++) and the module definition file (*source-file.m*).

For more information on module management behavior and influencing the preprocessor, see [Module Management Behavior](#) on page 17-8. For recommended naming conventions for C/C++ source files, see [Table 17-1](#) on page 17-1.

The preprocessor converts embedded SQL statements to C comments, followed by the appropriate CLI calls.

The C/C++ annotated source file consists of:

Header	Contains the declarations within the CLI functions and data structures.
Body	Contains the embedded SQL C/C++ source file translated into C/C++ statements. The preprocessor encloses each embedded SQL statement with C comment delimiters and follows the commented statement with a CLI call that invokes the executor at run time to execute the statement.
Trailer	Contains definitions required to complete the C/C++ source file. Definitions include the module version number, the creation timestamp (the operating system timestamp when the preprocessor was invoked), and the module name.

Header for Module Definition File

If you specify the `-m` or `-x` preprocessor option or set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable, the preprocessor

creates a module definition file in your current directory that contains embedded SQL statements. The preprocessor writes the header of the module definition file as:

```
MODULE module-name NAMES ARE ISO88591;
TIMESTAMP DEFINITION ( creation_timestamp );
source-file 'source-file location';
```

You can specify *module-name* by using the MODULE directive in your embedded SQL C/C++ program. For example:

```
EXEC SQL MODULE TX015.SQLPP.T0003N12;
```

The preprocessor translates this MODULE directive into:

```
MODULE TX015.SQLPP.T0003N12 NAMES ARE ISO88591;
TIMESTAMP DEFINITION (2110378403655251203)
SOURCE_FILE '/E/KINGPIN/usr/test/qalib/mxR2/cct0003/n12.ppp' ;
```

Otherwise, if you do not specify a MODULE directive, the preprocessor generates a system-supplied module name for you. See also the MODULE directive in the *SQL/MX Reference Manual*.

Trailer for Annotated-Source File

The *module-name* and the *creation_timestamp* correspond to these same elements in the trailer of the C/C++ source file. The SQL/MX compiler uses *module-name* to name the module file. It also writes the *creation_timestamp* into the module file. The C/C++ source file is then compiled and linked. When the resulting program file is executed and calls the SQL/MX executor, the preprocessor-generated CLI procedure calls pass the *module-name* and *creation_timestamp* to the executor. The executor uses the *module-name* to locate the corresponding module file. The *creation_timestamp* is used to ensure that the version of the executable program is synchronized with the version of the module file. This strategy prevents, for example, the executable program from being altered and rebuilt without rebuilding the module file. For more information, see [Understanding and Avoiding Some Common Run-Time Errors](#) on page 15-73.

The ISO88591 character set is the default character set for CHAR or VARCHAR data types for NonStop SQL/MX.

Procedures

After writing to the header of the module definition file, the preprocessor writes procedures for executing SQL statements. A procedure consists of a name, a formal argument list, and an SQL statement as the body of the procedure.

Each formal argument has a name and an SQL data type. The arguments are the host variables that occur in the SQL statement in the body of the procedure. The preprocessor writes the arguments in the same order as the first occurrence of the host variables, scanning from left to right, in the SQL statement. In some cases, the arguments are data structures that contain references to host variables. The host

variable references are stored in the same order in which they appear in the SQL statement.

OSS-Hosted SQL/MX C/C++ Preprocessor

You can compile and run an embedded SQL C/C++ program in the OSS environment on a NonStop system. Although you cannot compile and run such a program in the Guardian environment, you can use an OSS pass-through command in the Guardian environment. For instructions on using the Windows-hosted SQL/MX C/C++ preprocessor, see [Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-26. For instructions on using the OSS pass-through command to execute the preprocessor in the Guardian environment, see [Building SQL/MX Guardian Applications in the Guardian Environment](#) on page 15-67.

The OSS-hosted SQL/MX C/C++ preprocessor (`mxsqlc`) is installed in the `/usr/tandem/sqlmx/bin` directory in the OSS environment. You can use the `c89` utility to preprocess embedded SQL C/C++ programs, compile C/C++ and run the SQL/MX compiler, and then link the C/C++ program. For more information, see [c89 Utility: Using One Command for All Compilation Steps](#) on page 15-44.

Syntax for the OSS-Hosted SQL/MX C/C++ Preprocessor

```

mxsqlc sql-file
[ -c output-file ]
[ -m module-def-file ]
[ -e ]
[ -n ]
[ -a ]
[ -l list-file ]
[ -p ]
[ -o ]
[ -t timestamp ]
[ -d flag[=value]]
[ -h ]
[ -i pathname ]
[ -x ]
[ -X ]
[ -g {moduleGroup[=module-group-specification-string]
      |moduleTableSet[=module-tableset-specification-
        string]
      |moduleVersion[=module-version-specification-
        string]
      |moduleCatalog[=module-catalog-name]
      |moduleSchema[=module-schema-name]
      } ]
[ -Q { [invokeCatalog=catalog-name]
      | [invokeSchema=schema-name]
      } ]
[ -I ]
[ -U {32 | 64} ]
[ -w {sqlcode | sqlstate | both} ]
[ -O ]
[ -f {CHAR_AS_ARRAY | CHAR_AS_STRING} ]

```

sql-file

is the name of the input C/C++ source file that contains embedded SQL statements.

-c output-file

is the name of the output preprocessed annotated source file that contains C/C++ statements and embedded SQL statements converted to comments. This file is the input for the C or C++ compiler (c89 utility). If you are preparing a C++ application, specify the name with the .C, .cc, .cpp, .cxx, or .c++ extension. The default is *source-file.c*, where *source-file* is the name of the SQL/MX C/C++ source file (for example, *sqlprog.sql*) without the file extension.

-m module-def-file

is the name of the output module definition file, which is the input file for the SQL/MX compiler. The default is *source-file.m*, where *source-file* is the

name of the SQL/MX C/C++ source file (for example, `sqlprog.sql`) without the file extension.

-e

generates CHARACTER data types for date-time data types. This behavior is compatible with NonStop SQL/MX Release 1.8. For more information, see [INVOKE and Date-Time and Interval Host Variables \(SQL/MX Release 1.8 Applications\)](#) on page 3-44.

-n

directs the preprocessor to automatically append a null terminator to all host variable character strings before they are fetched into. Using the -n option does have the potential to produce nonportable code. Moreover, if the -a option is used together with the -n option, the -n option has no effect on VARCHARs.

-a

specifies that VARCHARs are to be translated into structures that contain a length and character string. For details about using this option, see [Generating Structures Instead of Using Null-Terminated Strings](#) on page 3-21 and [Example: Using a Structure](#) on page 3-22.

This preprocessor option overrides the SQL/MX default VARCHAR, which is a string with a null terminator. Moreover, if the -a option is used together with the -n option, the -n option has no effect on VARCHARs.

-l *list-file*

is the name of the output list file that contains preprocessor error and warning messages. The default is *source-file.lst*, where *source-file* is the name of the SQL/MX C/C++ source file (for example, `sqlprog.sql`) without the file extension.

-p

turns off the automatic generation of `#line` directives in the C/C++ output file, disables source-level debugging, and shows the generated C/C++ code for debugging purposes.

-o

overrides the use of Tandem floating point and uses IEEE floating point instead for host variables. In addition, if used with invoked SQL/MP tables with a column of type REAL, this option causes the invoked structure to be of type DOUBLE. For more information, see [INVOKE and Floating-Point Host Variables](#) on page 3-45.

By default, the c89 compiler (TNS/E targeted compilation) defaults to IEEE_float and will invoke the -o option when it calls the preprocessor. If you want your program to use Tandem_float, use the `c89 -wtandem_float` option to compile the `mssqlc`-generated annotated source file.

`-t timestamp`

provides a creation timestamp that the preprocessor writes to the C/C++ annotated source file (and the module definition file if the `-x` or `-m` preprocessor option or the `SQLMX_PREPROCESSOR_VERSION=800` environment variable is used). The *timestamp* value overrides the operating system timestamp. The value of the timestamp must be in Julian format.

For example, you can specify the following timestamp value:

`-t 2012000000000000036`

The preprocessing timestamp of the generated code must match the preprocessing timestamp stored in the module. Use this option with caution and only when you need to change the source text of the embedded SQL program without SQL-compiling the generated code.

`-d flag[=value]`

specifies a flag macro for later use in the conditional compilation of the source file. *flag* specifies the name of the macro and must be a valid C identifier. *value* can be any integer value (positive or negative). You cannot put spaces around the equal sign if an optional *value* is supplied.

The use of this option corresponds, for example, to the `#define` directive that might be found in a source file (that is, `#define foo 1`, where 1 is the value assigned to `foo`). The value can then be tested in an `#if` directive during preprocessing.

You can specify the option more than once on the command line.

`-h`

enables the processing of files specified in the user `#include` directive regardless of their extension. The default action is to ignore these files.

`-i pathname`

specifies a directory path to be searched for a file specified in an `#include` directive. The source path is searched first.

You can specify this option for a maximum of 20 paths.

`[-U {32 | 64}]`

Specifies the data model of the application to be either 32-bit or 64-bit. If you do not specify the data model option while processing the embedded SQL source file, the preprocessor uses 32-bit as the default data model.

The options `-U 32` and `-U 64` are not valid for the SQL/MX COBOL preprocessor. If specified, the SQL/MX COBOL preprocessor returns the error “13011: <option> is an unknown command line option”.

Note. The applications preprocessed with -U 64 must be linked with YCLIDLL.

-x

directs the preprocessor to refrain from emitting embedded module definitions into the annotated output source file.

-X

instructs the preprocessor to use the precision and scale in SET DESCRIPTOR statement for the data associated with the dynamic parameter through VARIABLE_DATA.

```
-g {moduleGroup[=module-group-specification-string]
   {moduleTableSet[=module-tableset-specification-string]
   {moduleVersion[=module-version-specification-string]
   {moduleCatalog[=module-catalog-name]
   {moduleSchema[=module-schema-name]
   }
   }
   }
```

specifies the arguments for qualifying the name given to the compiled module file. If you use this option, you must supply at least one of the five module management attributes. If you want to specify more than one attribute, repeat the entire -g option for each attribute. These attribute values are used to qualify the name of the compiled module file. See [File Naming Conventions](#) on page 17-1.

To use the -g option, you must supply a value in conjunction with the moduleGroup, moduleTableSet, moduleVersion, moduleCatalog, or moduleSchema attribute. The value must immediately follow the equal sign, and the equal sign must immediately follow the attribute keyword. The value can use regular or delimited identifiers. (See the description of regular and delimited identifiers in the *SQL/MX Reference Manual*.) If you supply more than one value for any attribute, only the final value is used. For information on the length of the module name, see [Module Name Length](#) on page 17-12.

moduleGroup

sets the moduleGroup attribute to group an application's module files logically by sharing the same name prefix. The moduleGroup becomes embedded in the module file names as a common prefix and enables the use of OSS wild-card file specification patterns to manage the files. For more information, see [Grouping](#) on page 17-23. The maximum size for the moduleGroup attribute is 31 characters.

moduleTableset

sets the moduleTableSet attribute to use the module management targeting feature. You can create different sets of module files that can be used against different sets of tables. For more information, see [Specifying the search](#)

[locations of the module files](#) on page 17-13. The maximum size for the `moduleTableSet` attribute is 31 characters.

`moduleVersion`

sets the `moduleVersion` attribute to enable multiple versions of an application's module files to coexist while keeping the same `MODULE` directive in each version. For more information, see [Versioning](#) on page 17-21. The maximum size for the `moduleVersion` attribute is 31 characters.

`moduleCatalog`

sets the `moduleCatalog` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a catalog name. If the `moduleCatalog` option is not set, the preprocessor emits the output `MODULE` directive by using the default catalog naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleCatalog` attribute is 128 characters.

`moduleSchema`

sets the `moduleSchema` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a schema name. The `moduleSchema` can contain a catalog name. If the `moduleSchema` attribute is not used, the preprocessor emits the output `MODULE` directive by using the default schema naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleSchema` attribute is 128 characters.

```
[ -Q { [ invokeCatalog=catalog-name ]
      [ invokeSchema=schema-name ]
    } ]
```

specifies the catalog name and schema name qualifiers for objects inside the `invoke` clause. If you use this option, specify one of the attributes—`invokeCatalog` or `invokeSchema`. If you want to specify both the attributes, repeat the `-Q` option for each attribute.

`invokeCatalog`

sets the catalog for unqualified objects inside the `invoke` clause as *catalog-name*. If a catalog is specified using the Control Query Default Catalog or Declare Catalog, this attribute has no effect. The maximum size of the `invokeCatalog` attribute is 128 characters.

`invokeSchema`

sets the schema for unqualified objects inside the `invoke` clause as *schema-name*. If a schema is specified using the Control Query Default Schema or Declare Schema, this attribute has no effect. The maximum size of the `invokeSchema` attribute is 128 characters.

-I

processes the nested `#include` files.

-w

handles warnings for `SQLCODE` and `SQLSTATE` declarations.

`sqlcode`

issues a warning if `SQLCODE` is undeclared or not declared as `long SQLCODE` in the Declare section.

`sqlstate`

issues a warning if `SQLSTATE` is undeclared or not declared as `char SQLSTATE[6]` in the Declare section.

`both`

issues warnings if either or both `SQLCODE` and `SQLSTATE` are undeclared or not declared as `long SQLCODE` and `char SQLSTATE[6]` respectively in the Declare section.

-O

replaces Guardian `DEFINE` in the `#include` directive, in the OSS file format. The `DEFINES` are resolved only if the preprocessor option `-O` is specified.

-f

specifies whether to reduce size by one or not for the null terminator of the character type descriptor. The default value is `CHAR_AS_STRING`. The following rules apply:

`CHAR_AS_STRING` – reduces the size by one from the value specified in the variable declaration.

`CHAR_AS_ARRAY` – retains the size specified in the variable declaration.

Example—mxsqlc

Run the SQL/MX C/C++ preprocessor using the `mxsqlc` command. This C++ example creates an annotated source file and module definition file:

```
mxsqlc sqlprog.ecpp -c sqlprog.cpp -x -m sqlprog.m -p \
-g moduleGroup=INVENTORY -g moduleVersion=V2
```

This C++ example creates a self-contained, annotated output source file that contains an embedded module definition:

```
mssqlc sqlprog.ecpp -c sqlprog.cpp -g moduleTableSet=TEST1
```

This C++ example creates an annotated source file and module definition file for 64-bit compilation.

```
Mssqlc sqlprog.ecpp -c sqlprog.cpp -x -m sqlprog.m -p -U 64
```

Windows-Hosted SQL/MX C/C++ Preprocessor

The Windows hosted SQL/MX C/C++ preprocessor is a DLL file named `mssqlcnt.dll` and is accompanied by a DLL loader named `mssqlc.exe`. These files are installed in the `C:\Program Files\HP SQL-MX C Preprocessors` directory. Use either the command shell or the Korn shell to run the preprocessor with the `RUN` command. You can also use ETK to build a C or C++ program. For more information, see [Building a C/C++ Program With Embedded SQL Statements on Windows](#) on page 15-55.

You can install multiple versions of the SQL/MX C/C++ preprocessors. The environment variable `MXSQLC` enables you to select a particular version of the SQL/MX C/C++ preprocessor for C/C++ compilations.

For example, to select the SQL/MX C/C++ preprocessor in the `C:\PROGRA~1\HPSQL--~1\` directory, set `MXSQLC` from the Windows command line:

```
set MXSQLC=C:\PROGRA~1\HPSQL--~1\mssqlcnt.dll
```

You can also set the environment variable in the Windows system properties. If multiple versions of the SQL/MX C/C++ preprocessors are installed and if `MXSQLC` is not set, the latest version of the SQL/MX C/C++ preprocessor installed on the system is used for compilations, by default.

Note. On systems running H06.14 RVU and later, the SQL/MX C/C++ compilations select the latest version of the SQL/MX C/C++ preprocessors installed on the system.

On systems running H06.13 RVU and earlier, the SQL/MX C/C++ compilations use the earliest version of the SQL/MX C/C++ preprocessor installed on the system.

If you use `INVOKE`, `MXCS` must be installed on your operating system to provide the necessary communication between your client workstation and the server. For more information on how to install `MXCS`, see the *SQL/MX Connectivity Service Administrative Command Reference*. In addition, you must install the HP NonStop ODBC/MX driver for Windows. For installation information, see the *ODBC/MX Driver for Windows Manual*.

The SQL/MX C/C++ preprocessor 2.3 can be invoked by a user or by a cross compiler, or by an Integrated Development Environment (IDE). For each scenario, the SQL preprocessor invoked is:

- A user calls the SQL preprocessor to preprocess a source program. The SQL preprocessor uses the header files and libraries from the SQL preprocessor installation directory.
- The SQL preprocessor is invoked by c89. c89 uses the SQL preprocessor version defined by `MXSQLC`. The SQL preprocessor uses the libraries and header files related to that version.

If `MXSQLC` is not set, the cross compiler invokes the latest version of the SQL/MX C/C++ preprocessor installed on the system.

- An IDE is used. The IDE invokes c89, which uses `MXSQLC` to select an alternative version. Some of the IDEs are:
 - Enterprise Tool Kit (ETK)—plug-in for Microsoft Visual studio
The environment variable `MXSQLC` must be set before starting ETK.
 - Enterprise Plugins for Eclipse (EPE)—plug-in for Eclipse
When Eclipse is used, `MXSQLC` is set by EPE based on the value of the preprocessor installation location.

Syntax for the Windows-Hosted SQL/MX C/C++ Preprocessor

```

mxsqlc sql-file
[ -c output-file ]
[ -m module-def-file ]
[ -e ]
[ -n ]
[ -a ]
[ -l list-file ]
[ -p ]
[ -o ]
[ -t timestamp ]
[ -d flag[=value]]
[ -s system-name or IP-address ]
[ -r ODBC-listener ]
[ -y NSK-username ]
[ -z NSK-password ]
[ -h ]
[ -i pathname ]
[ -x ]
[ -X ]
[ -g {moduleGroup[=module-group-specification-string]
      |moduleTableSet[=module-tableset-specification-
      string]
      |moduleVersion[=module-version-specification-
      string]
      |moduleCatalog[=module-catalog-name]
      |moduleSchema[=module-schema-name]
      } ]
[ -Q { [invokeCatalog=catalog-name]
      | [invokeSchema=schema-name]
      } ]
[ -I ]
[ -U {32 | 64} ]
[ -w {sqlcode | sqlstate | both } ]
[ -f {CHAR_AS_ARRAY | CHAR_AS_STRING} ]

```

sql-file

is the name of the input C/C++ source file that contains embedded SQL statements.

-c output-file

is the name of the output preprocessed annotated source file that contains C/C++ statements and embedded SQL statements converted to comments. This file is the input file for the C or C++ compiler. If you are preparing a C++ application, specify the name with the .C, .cc, .cpp, .cxx, or .c++ extension. The default is *source-file.c*, where *source-file* is the name of the SQL/MX C/C++ source file (for example, *sqlprog.sql*) without the file extension.

`-m module-def-file`

is the name of the output module definition file, which is the input file for the SQL/MX compiler. The default is *source-file.m*, where *source-file* is the name of the SQL/MX C/C++ source file (for example, *sqlprog.sql*) without the file extension.

`-e`

generates CHARACTER data types for date-time data types. This behavior is compatible with NonStop SQL/MX Release 1.8. For more information, see [INVOKE and Date-Time and Interval Host Variables \(SQL/MX Release 1.8 Applications\)](#) on page 3-44.

`-n`

directs the preprocessor to automatically append a null terminator to all host variable character strings before they are fetched into. Moreover, if the `-a` option is used together with the `-n` option, the `-n` option has no effect on VARCHARs. Using the `-n` option does have the potential to produce nonportable code.

`-a`

specifies that VARCHARs are to be translated into structures that contain a length and character string. For details about using this option, see [Generating Structures Instead of Using Null-Terminated Strings](#) on page 3-21 and [Example: Using a Structure](#) on page 3-22.

This preprocessor option overrides the SQL/MX default VARCHAR, which is a string with a null terminator. Moreover, if the `-a` option is used together with the `-n` option, the `-n` option has no effect on VARCHARs.

`-l list-file`

is the name of the output list file that contains preprocessor error and warning messages. The default is *source-file.lst*, where *source-file* is the name of the SQL/MX C/C++ source file (for example, *sqlprog.sql*) without the file extension.

`-p`

turns off the automatic generation of `#line` directives in the C/C++ output file, disables source-level debugging, and shows the generated C/C++ code for debugging purposes.

`-o`

overrides the use of Tandem floating point and uses IEEE floating point instead for host variables. In addition, if used with invoked SQL/MP tables with a column of type REAL, this option causes the invoked structure to be of type DOUBLE. For more information, see [INVOKE and Floating-Point Host Variables](#) on page 3-45.

By default, the c89 compiler (TNS/E targeted compilation) defaults to IEEE_float and will invoke the -o option when it calls the preprocessor. If you want your program to use Tandem_float, use the c89 -wtandem_float option to compile the mxsqlc-generated annotated source file.

-t *timestamp*

provides a creation timestamp that the preprocessor writes to the C/C++ annotated source file (and the module definition file if the -x or -m preprocessor option or the SQLMX_PREPROCESSOR_VERSION=800 environment variable is used). The *timestamp* value overrides the operating system timestamp. The value of the timestamp must be in Julian format.

For example, you can specify the following timestamp value:

-t 2012000000000000036

The preprocessing timestamp of the generated code must match the preprocessing timestamp stored in the module. Use this option with caution and only when you need to change the source text of the embedded SQL program without SQL-compiling the generated code.

-d *flag[=value]*

is a flag macro for later use in the conditional compilation of the source file. The *flag* specifies the name of the macro and must be a valid C identifier. The *value* can be any integer value (positive or negative). You cannot put spaces around the equal sign if an optional *value* is supplied.

The use of this option corresponds, for example, to the #define directive that might be found in a source file (that is, #define foo 1, where 1 is the value assigned to foo). The value can then be tested in a #if directive during preprocessing.

Because the preprocessor does not process #include files, you must use this option to define any macros that are typically defined in #include files and that affect the conditional processing of the source file. You can specify the option more than once on the command line.

-s *system-name or IP-address*

is the node name or IP address of the NonStop system where the tables are found by INVOKE. This option is required if you use INVOKE.

-r *ODBC-listener*

is the NonStop system port to connect to for the ODBC listener process. The default port for the Association server is 18650.

`-y NSK-username`

is the Guardian user name with access to the tables that INVOKE reads. This option is required if you use INVOKE.

`-z NSK-password`

is the password for the user name for the NonStop system. This option is required if you use INVOKE.

`-h`

enables the processing of files specified in the user `#include` directive regardless of their extension. The default action is to ignore these files.

`-i pathname`

specifies a directory path to be searched for a file specified in an `#include` directive. The source path is searched first.

You can specify this option for a maximum of 20 paths.

[`-U {32 | 64}`]

Specifies the data model of the application to be either 32-bit or 64-bit. If you do not specify the data model option while processing the embedded SQL source file, the preprocessor uses 32-bit as the default data model.

The options `-U 32` and `-U 64` are not valid for the SQL/MX COBOL preprocessor. If specified, the SQL/MX COBOL preprocessor returns the error “13011: <option> is an unknown command line option”.

Note. The applications preprocessed with `-U 64` must be linked with YCLIDLL.

`-x`

directs the preprocessor to refrain from emitting embedded module definitions into the annotated output source file.

`-X`

instructs the preprocessor to use the precision and scale in SET DESCRIPTOR statement for the data associated with the dynamic parameter through VARIABLE_DATA.

```
-g { moduleGroup[=module-group-specification-string]
    moduleTableSet[=module-tableset-specification-string]
    moduleVersion[=module-version-specification-string]
    moduleCatalog[=module-catalog-name]
    moduleSchema[=module-schema-name]
}
```

specifies the arguments for qualifying the name given to the compiled module file. If you use this option, you must supply at least one of the five module management attributes. If you want to specify more than one attribute, repeat the entire `-g` option for each attribute. These attribute values are used to qualify the name of the compiled module file. For more information, see [File Naming Conventions](#) on page 17-1.

To use the `-g` option, you must supply a value in conjunction with the `moduleGroup`, `moduleTableSet`, `moduleVersion`, `moduleCatalog`, or `moduleSchema` attribute. The value must immediately follow the equal sign, and the equal sign must immediately follow the attribute keyword. The value can use regular or delimited identifiers. (See the description of regular and delimited identifiers in the *SQL/MX Reference Manual*.) If you supply more than one value for any attribute, only the final value is used. For information on the length of the module name, see [Module Name Length](#) on page 17-12.

`moduleGroup`

sets the `moduleGroup` attribute to group an application's module files logically by sharing the same name prefix. The `moduleGroup` becomes embedded in the module file names as a common prefix and enables the use of OSS wild-card file specification patterns to manage the files. For more information, see [Grouping](#) on page 17-23. The maximum size for the `moduleGroup` attribute is 31 characters.

`moduleTableset`

sets the `moduleTableSet` attribute to use the module management targeting feature. You can create different sets of module files that can be used against different sets of tables. For more information, see [Specifying the search locations of the module files](#) on page 17-13. The maximum size for the `moduleTableSet` attribute is 31 characters.

`moduleVersion`

sets the `moduleVersion` attribute to enable multiple versions of an application's module files to coexist while keeping the same `MODULE` directive in each version. For more information, see [Versioning](#) on page 17-21. The maximum size for the `moduleVersion` attribute is 31 characters.

`moduleCatalog`

sets the `moduleCatalog` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a catalog name. If

the `moduleCatalog` option is not set, the preprocessor emits the output `MODULE` directive by using the default catalog naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleCatalog` attribute is 128 characters.

`moduleSchema`

sets the `moduleSchema` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a schema name. The `moduleSchema` can contain a catalog name. If the `moduleSchema` attribute is not used, the preprocessor emits the output `MODULE` directive by using the default schema naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleSchema` attribute is 128 characters.

```
[ -Q { [invokeCatalog=catalog-name]
      | [invokeSchema=schema-name]
    } ]
```

specifies the catalog name and schema name qualifiers for objects inside the `invoke` clause. If you use this option, specify one of the attributes—`invokeCatalog` or `invokeSchema`. If you want to specify both the attributes, repeat the `-Q` option for each attribute.

`invokeCatalog`

sets the catalog for unqualified objects inside the `invoke` clause as *catalog-name*. If a catalog is specified using the Control Query Default Catalog or Declare Catalog, this attribute has no effect. The maximum size of the `invokeCatalog` attribute is 128 characters.

`invokeSchema`

sets the schema for unqualified objects inside the `invoke` clause as *schema-name*. If a schema is specified using the Control Query Default Schema or Declare Schema, this attribute has no effect. The maximum size of the `invokeSchema` attribute is 128 characters.

`-I`

processes the nested `#include` files.

`-w`

handles warnings for `SQLCODE` and `SQLSTATE` declarations.

`sqlcode`

issues a warning if `SQLCODE` is undeclared or not declared as `long SQLCODE` in the Declare section.

`sqlstate`

issues a warning if `SQLSTATE` is undeclared or not declared as `char SQLSTATE[6]` in the Declare section.

`both`

issues warnings if either or both `SQLCODE` and `SQLSTATE` are undeclared or not declared as `long SQLCODE` and `char SQLSTATE[6]` respectively in the Declare section.

`-f`

specifies whether to reduce size by one or not for the null terminator of the character type descriptor. The default value is `CHAR_AS_STRING`. The following rules apply:

`CHAR_AS_STRING` – reduces the size by one from the value specified in the variable declaration.

`CHAR_AS_ARRAY` – retains the size specified in the variable declaration.

Example—mxsqlc

Run the SQL/MX C/C++ preprocessor using the `mxsqlc` command. This C++ example creates an annotated source file and module definition file:

```
mxsqlc sqlprog.ecpp -c sqlprog.cpp -x -m sqlprog.m -p \
-g moduleGroup=INVENTORY -g moduleVersion=V2
```

This C++ example creates a single-file annotated output source file that contains an embedded module definition:

```
mxsqlc sqlprog.ecpp -c sqlprog.cpp -g moduleTableSet=TEST1
```

This C++ example creates an annotated source file and module definition file for 64-bit compilation.

```
Mxsqlc sqlprog.ecpp -c sqlprog.cpp -x -m sqlprog.m -p -U 64
```

Running the C/C++ Compiler and Linker

The HP NonStop C/C++ compilers translate source code into machine language that is specific to a particular NonStop system architecture. The type of C/C++ compiler that you use to compile your SQL/MX program determines the NonStop system and environment where you can run the program.

Note. TNS/R native compilation tools are available on systems running H06.05 or later RVUs.

[Table 15-1](#) on page 15-35 lists the C/C++ compilers, the environments where you can run the compilers, and the environments where you can run the compiled programs.

Table 15-1. HP NonStop C/C++ Compilers for Embedded SQL/MX Programs

Compiler	Compiler Operating Environment	Program Execution Environment
TNS/E native compilers:		
● Native C cross compiler for TNS/E*	Windows environment on a PC connected to a NonStop system running an H-series RVU	OSS or Guardian environment on a NonStop system running an H-series RVU
● c89	OSS environment on a NonStop system running an H-series RVU	OSS or Guardian environment on a NonStop system running an H-series RVU
● CCOMP (C)	Guardian environment on a NonStop system running an H-series RVU	Guardian or OSS environment on a NonStop system running an H-series RVU
● CPPCOMP (C++)	Guardian environment on a NonStop system running an H-series RVU	Guardian or OSS environment on a NonStop system running an H-series RVU
TNS/R native compilers:		
● Native C cross compiler for TNS/R*	Windows environment on a PC connected to a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	OSS or Guardian environment on a NonStop system running a G-series RVU
● c89	OSS environment on a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	OSS or Guardian environment on a NonStop system running a G-series RVU
● NMC (C)	Guardian environment on a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	Guardian or OSS environment on a NonStop system running a G-series RVU
● NMCPLUS (C++)	Guardian environment on a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	Guardian or OSS environment on a NonStop system running a G-series RVU
* The native C cross compilers can be run from ETK or from the PC command line.		

On Windows, you can run the C/C++ compiler and native object file linker from ETK, or you can use the command-line cross compiler `c89` and the linker (`eld` or `nld`). For details on syntax and using the C/C++ cross compiler with ETK, see the help file *Using Command-Line Cross Compilers on Windows*, which is included with ETK.

Note. The default C++ run-time library for CPPCOMP and NMCPLUS is `version3`. You can use either `version3` or `version2` when you issue your C++ compiler command.

To run the C/C++ compilers, see the *C/C++ Programmer's Guide*. To run the `eld` linker, see the *eld Manual*. To run the `nld` linker, see the *nld Manual*. For more information on the `c89` utility, see [c89 Utility: Using One Command for All Compilation Steps](#) on page 15-44, the OSS reference pages, or the *Open System Services Shell and Utilities Reference Manual*.

Running the SQL/MX Compiler

The SQL/MX compiler compiles and optimizes static and dynamic SQL statements for subsequent execution by the SQL/MX executor and performs these functions:

- Expands SQL object names by using the current default settings
- Expands view definitions
- Performs type checking for C/C++ and SQL data types
- Checks SQL object references to verify their existence
- Determines an optimized execution plan and access path for each DML statement if the SQL objects in the statement are present at compile time
- Generates executable code for the execution plans (if the SQL objects in the statement are present at compile time) and creates a module in the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.
- Generates a list of SQL statements in the program file, including messages
- Returns a completion code indicating the outcome of the compilation

The SQL/MX compiler is an OSS program installed in the Guardian `$SYSTEM.SYSTEM` subvolume (`/G/system/system/` in the OSS environment). You must run the compiler in the OSS environment. It does not run as a Guardian process.

You must explicitly invoke the SQL/MX compiler to compile static SQL statements. At run time, the SQL/MX executor also invokes the compiler to compile dynamic SQL statements and to recompile any static SQL statements that refer to database objects that have changed and that affect the SQL statement's execution plan.

If your program accesses a table that has changed since the last static compilation, you should statically recompile the program to improve performance. Otherwise, NonStop SQL/MX dynamically recompiles the program before each execution.

Compiling Embedded Module Definitions

To compile one or more of the modules of an embedded SQL/MX application executable, use the `mxCompileUserModule` utility on the object file created by the C/C++ compiler or on the executable file produced by the linker.

If you have a combination of module definition files and applications that contain embedded module definitions, use `mxCompileUserModule` to SQL compile the self-contained object files containing embedded module definitions, and use `mxcmp` to SQL compile the application's separate module definition files. For more information, see [Compiling a Module Definition File](#) on page 15-42 and the example [Building and Deploying a C Application With Embedded Module Definitions and Module Definition Files](#) on page 15-58.

Command-Line Syntax

To invoke `mxCompileUserModule`, at an OSS prompt, enter:

```
mxCompileUserModule { { [-e] [-v] [-g {moduleGlobal |
moduleLocal}]
[-d compiler-attribute-name=compiler-attribute-value]... } |
-m } Application-file ["{"module-name [, module-name]...""}"

Module-name is:
[[Catalog.]Schema.]Module [MODULEGROUP=group]
[MODULETABLESET=target] [MODULEVERSION=version]
```

-e

directs `mxCompileUserModule` to generate a warning rather than an error if a table or class MAP DEFINE in an SQL statement does not exist during explicit SQL/MX compilation. To find errors in a program during explicit SQL/MX compilation, omit the `-e` option.

If you are using late name resolution and want to use a table or DEFINE that does not exist during explicit SQL/MX compilation, include the `-e` option. Then at run time, the SQL/MX executor automatically recompiles the SQL statement from the statement's source in the module by using the run-time version of the table.

-v

directs `mxCompileUserModule` to display summary information in addition to error and warning messages for the compilation. For example, use this option to verify the default settings of the SQL/MX compiler.

-m

directs `mxCompileUserModule` to display the list of module files associated with the application file.

Note.

- If `-m` option is specified, other command line options are ignored.
 - The OSS tool `mxCompileUserModule` with the `-m` option does not display the module files associated with the DLLs loaded by the embedded SQL executable.
 - `mxCompileUserModule` with the `-m` option does not display the list of module names associated with an application file, if the modules are generated from a source SQL file, using the `-x` preprocessor option, or if the environment variable, `SQLMX_PREPROCESSOR_VERSION` is set to 800.
-

-g moduleGlobal

specifies that the module is placed globally in the `/usr/tandem/sqlmx/USERMODULES` directory.

-g moduleLocal[=<OSSdir>]

directs `mxCompileUserModule` to place the module in the OSS directory. The OSS directory can be either a Guardian or OSS location in the OSS format. If the OSS directory is omitted, the module is created in the current directory. The following rules related to the OSS directory apply:

- The OSS directory must exist and be accessible.
- The directory must not be a remote directory in an Expand network.
- The OSS directory must not exceed 1024 characters.

If these conditions are not met, an error is generated, and no module is created.

If you do not specify `-g moduleLocal[=<OSSdir>]` but set `MXCMP_PLACES_LOCAL_MODULES ON`, you must be in the same directory as the application executable when you invoke `mxCompileUserModule`. Otherwise, `mxCompileUserModule` writes the module in the current directory, and you will need to move the module to the global `USERMODULES` directory or co-locate the module with its application. For more information, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

-d *compiler-attribute-name=compiler-attribute-value*

specifies default attribute settings for compilation and existing settings for the module name. The module name settings are:

`modulecatalog=cat`
`moduleschema=sch`

```
modulegroup=grp
moduletableset=tgt
moduleversion=ver
```

The `-d modulecatalog`, `moduleschema`, `modulegroup`, `moduletableset`, and `moduleversion` options are similar to the `mxsqlc -g modulecatalog`, `moduleschema`, `modulegroup`, `moduletableset`, and `moduleversion` options because you use them to externally qualify simple module names. These options are not CONTROL QUERY DEFAULT settings (however, all other `-d attr=value` pairs are). In addition, there is no default value for `-d modulecatalog` or `-d moduleschema`.

The module name settings must match the module management options you specified during preprocessing. See [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8.

The default attribute settings for compilation override settings in the `SYSTEM_DEFAULTS` table but do not override the object name qualification or the settings of embedded CONTROL QUERY DEFAULT, DECLARE, or SET statements, which are in the input source file. For more information, see the `SYSTEM_DEFAULTS` table in the *SQL/MX Reference Manual*.

The OSS shell is used to invoke `mxCompileUserModule`, which in turn uses the OSS shell to invoke `mxcmp`. Consequently, you must adjust the syntax for setting CONTROL QUERY DEFAULT attribute values for `MP_SYSTEM` and `MP_VOLUME`. The OSS shell performs command/parameter substitution and allows a `\` (backslash) to quote special characters such as `$`.

This example shows how to set `MP_SYSTEM` and `MP_VOLUME` as `mxCompileUserModule` command-line options:

```
to get MP_SYSTEM=\KINGPIN    --> use -d MP_SYSTEM=\\KINGPIN
to get MP_VOLUME=$TX012      --> use -d MP_VOLUME=\\$TX012
```

application-file

is the OSS path name of an object file that contains embedded module definitions. The OSS directory:

- Must exist and be accessible. Otherwise, an error is returned, and no module is created.
- Must not specify a Guardian subvolume (`/G/...`) or a remote directory in an Expand network (`/E/...`).
- Must not exceed 1024 characters.

module-name

is the fully qualified name of an embedded module definition. This option names the generated module that is written to the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL

location(s) or to the global `/usr/tandem/sqlmx/USERMODULES` directory. For more information, see [Module Management Naming](#) on page 17-9.

Each *module-name* consists of:

```
[[catalog.]schema.]module[MODULEGROUP=group]  
[MODULETABLESET=target][MODULEVERSION=version]
```

If *catalog* and *schema* are omitted, their default value settings can be supplied with `-d MODULECATALOG=catalog` or `-d MODULESCHEMA=schema`. If `MODULEGROUP`, `MODULETABLESET`, or `MODULEVERSION` is omitted, the default setting can be supplied with `-d MODULEGROUP=group`, `-d MODULETABLESET=target`, or `-d MODULEVERSION=version`.

If no module name is specified, `mxCompileUserModule` operates on all embedded module definitions of *application-file*. Otherwise, each *module-name* is the fully qualified three-part name of an embedded module definition in *application-file*, and `mxCompileUserModule` operates only on the named embedded module definitions.

In summary, modules can be named as:

- A fully qualified delimited module name, such as
`cat.sch.\"GRP^MODULE^TGT^VER\"`
- A qualified module name followed by module specification strings, such as
`cat.sch.module MODULEGROUP=grp MODULETABLESET=tgt
MODULEVERSION=ver`
- A simple, unqualified module name (for example, `mod`), with the catalog, schema, group, table set, or version specified as `-d` compiler attributes

You can run `mxCompileUserModule` more than once.

`mxCompileUserModule` extracts the *application-file*'s selected module definitions. For each selected module definition *m*, `mxCompileUserModule` passes *m* to `mxcmp` for SQL compilation. Each compilation of a selected module definition either succeeds or fails just like any `mxcmp` invocation. An `mxcmp` compilation failure does not affect preceding or following `mxcmp` invocations. In particular, an `mxcmp` compilation failure does not prevent `mxCompileUserModule` from proceeding with the `mxcmp` compilation of the next selected module definition.

Examples—mxCompileUserModule

- This command compiles the embedded module definition:
`mxCompileUserModule sqlprog.exe`
- This command places the module file in the same OSS directory as the application executable:
`mxCompileUserModule -g moduleLocal sqlprog.o`

- These settings affect statement recompilation at execution time:

```
mxCompileUserModule -d AUTOMATIC_RECOMPILATION=ON \
-d SIMILARITY_CHECK=ON sqlprog.o
```

- The following command compiles the embedded module definition and places the module file in the user-specified OSS location, /usr/mymodules:

```
mxcompileusermodule -g moduleLocal=/usr/mymodules sqlprog.exe
```

- The following command compiles the embedded module definition and places the module file in the user-specified Guardian location, /G/data01/mymod:

```
mxcompileusermodule -g moduleLocal=/G/data01/mymod
sqlprog.exe
```

- The following command displays the list of module files associated with the file, CAT.SCH.TEST.EXE:

```
mxCompileUserModule -m CAT.SCH.TEST.EXE

List Of Modules:
CAT.SCH.TEST.EXE

1 module found, 0 modules extracted
0 mxcmp invocations: 0 succeeded, 0 failed
```

MXCMP Environment Variable

To specify an alternate location of the SQL/MX compiler (MXCMP) instead of the default location of /G/system/system/mxcmp, use the MXCMP environment variable. This environment variable is used by c89 and the mxCompileUserModule utility and enables you to direct c89 or mxCompileUserModule to use another version of the MXCMP executable.

To set the MXCMP environment variable, enter this command at an OSS prompt before invoking the c89 or mxCompileUserModule utility:

```
export MXCMP="/G/usr/mydir/mxcmp"
```

For more information, see the *Open System Services Shell and Utilities Reference Manual*.

MXCMPUM Environment Variable

To specify an alternate location of the compiler utility (mxCompileUserModule) instead of the default location of /usr/tandem/sqlmx/bin/mxCompileUserModule, use the MXCMPUM environment variable. This environment variable is used by the c89 utility and enables you to direct c89 to use another version of the mxCompileUserModule utility.

To set the MXCMPUM environment variable, enter this command at an OSS prompt before invoking the c89 utility:

```
export MXCMPUM="/G/usr/mydir/mxCompileUserModule"
```

For more information, see the *Open System Services Shell and Utilities Reference Manual*.

Compiling a Module Definition File

Embedded SQL application source files preprocessed with the `-x` and `-m` options or that set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable continue to generate module definition files as is done in SQL/MX Release 1.8 and previous releases.

To compile a module definition file, use the SQL/MX compiler `mxcmp` command on the module definition (`.m`) file. The SQL/MX compiler places a compiled user module file in the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.

Command-Line Syntax

To invoke the SQL/MX compiler, at an OSS prompt, enter:

```
mxcmp  [ -e ] [ -v ]
        [ -g {moduleGlobal|moduleLocal[=OSSdir]} ]
        [ -d compiler-attribute-name=compiler-attribute-value ] ...
        module-definition-file
```

`-e`

directs `mxcmp` to generate a warning rather than an error if a table or class MAP DEFINE in an SQL statement does not exist during explicit SQL/MX compilation. To find errors in a program during explicit SQL/MX compilation, omit the `-e` option.

If you are using late name resolution and want to use a table or DEFINE that does not exist during explicit SQL/MX compilation, include the `-e` option. Then at run time, the SQL/MX executor automatically recompiles the SQL statement from the statement's source in the module by using the run-time version of the table.

`-v`

directs `mxcmp` to display summary information in addition to error and warning messages for the compilation.

`-g moduleGlobal`

specifies that the module is placed globally in the `/usr/tandem/sqlmx/USERMODULES` directory.

`-g moduleLocal[=OSSdir]`

directs `mxcmp` to place the module in the named OSS directory. The OSS directory can be either a Guardian or an OSS location in the OSS format. If the OSS

directory is omitted, the module is created in the current directory. The following rules related to the OSS directory apply:

- The OSS directory must exist and be accessible.
- The OSS directory must not be a remote directory in an Expand network.
- The OSS directory must not exceed 1024 characters.

If these conditions are not met, an error is generated, and no module is created.

If you do not specify `-g moduleLocal=OSSdir`, but set `MXCMP_PLACES_LOCAL_MODULES ON`, you must be in the same directory as the application executable when you invoke `mxcmp`. Otherwise, `mxcmp` writes the module in the current directory, and you will need to move the module to the global `USERMODULES` directory or co-locate the module with its application. For more information, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

`-d compiler-attribute-name=compiler-attribute-value`

specifies default attribute settings for compilation. The default attribute settings for compilation override settings in the `SYSTEM_DEFAULTS` table but do not override the object name qualification or the settings of embedded `CONTROL QUERY DEFAULT`, `DECLARE`, or `SET` statements, which are in the input source file. For more information, see the `SYSTEM_DEFAULTS` table in the *SQL/MX Reference Manual*.

The OSS shell is used to invoke `mxcmp`. Consequently, you must adjust the syntax for setting `CONTROL QUERY DEFAULT` attribute values for `MP_SYSTEM` and `MP_VOLUME`. The OSS shell performs command/parameter substitution and allows a `\` (backslash) to quote special characters such as `$`.

This example shows how to set `MP_SYSTEM` and `MP_VOLUME` as `mxcmp` command-line options:

```
to get MP_SYSTEM=\KINGPIN    --> use -d MP_SYSTEM=\\KINGPIN
to get MP_VOLUME=$TX012      --> use -d MP_VOLUME=\\$TX012
```

You must use a pair of backslashes when specifying the value for `MP_SYSTEM` and one for `MP_VOLUME`.

module-definition-file

is the name of the input module definition file (`.m`) that was generated by the C/C++ preprocessor (`mxsqlc`).

The static SQL/MX compiler provides backward compatible behavior. If the `SQLMX_PREPROCESSOR_VERSION` environment variable is set to 800, `mxcmp` behaves just like SQL/MX Release 1.8. Otherwise, `mxcmp` supports all SQL/MX Release 2.x features and command-line options. For more information, see [Influencing Module Management Behavior](#) on page 17-9.

Example—mxcmp

The following command compiles the module definition and places module file in the user specified OSS location, `/usr/mymodules`:

```
mxcmp -g moduleLocal=/usr/mymodules sqlprog.m
```

The following command compiles the module definition and places module file in the user specified Guardian location, `/G/data01/mymod`:

```
mxcmp -g moduleLocal=/G/data01/mymod sqlprog.m
```

c89 Utility: Using One Command for All Compilation Steps

In the OSS environment, the `c89` utility provides the interface to C/C++ compilation components, including the SQL/MX C/C++ preprocessor, the native C and C++ compilers, native C run-time library, and the native object file linker (`eld` or `nld`). `nld` is available on systems running H06.05 or later RVUs. The `c89` utility enables you to build an embedded SQL C or C++ program in a single command. You can also use `c89` utility options individually: for example, to run the SQL/MX compiler after preprocessing.

In the Windows environment, the `c89` utility is bundled with ETK. For details on syntax and use, see the help file *Using Command-Line Cross Compilers on Windows*, which is included with ETK. In addition, the *Open System Services Shell and Utilities Reference Manual* contains a listing of all `c89` utility options.

The `c89` utility is installed in the `/usr/bin` directory.

c89 Utility Options for SQL/MX

`-WDname [=value]`

Specifies a macro that sends class MAP `DEFINES` for the SQL/MX preprocessor to use for conditional compilation during the SQL/MX preprocessing step.

`-Wsqlmx[={ "args" | args }]`

Invokes the SQL/MX preprocessor prior to invoking the C or C++ compiler. Cannot be specified with `-Wsql`, `-Wsqlcomp`, or `-Wmigrate`. Ignored if any options that prevent compilation are specified: `-E`, `-WC`, `-WH`, `-WM`, `-WP`, `-Wsyntax`.

You can use one or more of the *args*, separated by commas without space between them.

`noline`

Prevents the SQL/MX preprocessor from generating `#line` directives in the preprocessed C/C++ annotated source it creates.

`listing`

Directs the SQL/MX preprocessor to write its diagnostic messages to a file named *file.eL* (where *file* is the name of the primary source file).

`preprocess_only`

Runs the SQL/MX preprocessor only.

`process_includes`

Processes one level of `#include` files.

`noansi_varchars`

Directs the SQL/MX preprocessor to turn off generation of ANSI `varchar` data.

`null_terminate`

Automatically appends a null terminator to all host variable character strings before they are fetched into. Moreover, if the `-a` option is used together with the `-n` option, the `-n` option has no effect on `VARCHARs`. The `-n` option has the potential to produce nonportable code.

`refrain_r2`

Directs the SQL/MX preprocessor to refrain from embedding module definitions in the annotated source file and to use a module definition file.

`-Wtandem_float`

For TNS/E targeted compilations, overrides `IEEE_float` and uses Tandem floating-point format. For correct and meaningful floating-point values in embedded programs, this option should be used only if the `mxsqlc -o` option is not used.

`-Wsqlmxadd=["args" | arg]`

Passes valid preprocessor commands (*args*) to the SQL/MX preprocessor without change or validation. The preprocessor validates the syntax.

`-Wtimestamp=value`

Passes a creation timestamp to the SQL/MX preprocessor. Ignored if `-Wsqlmx` is not specified. If set more than once, only the last occurrence takes effect. `c89` does not validate *value*. Validation is left to the preprocessor. For details about the form of *value*, see [-t timestamp](#) on page 15-22.

`-Wmxcmp[={ "args" | args }]`

Invokes the SQL/MX compiler. If compiling with embedded module definitions, invokes `mxCompileUserModule`. If compiling with separate module definition files, invokes `mxcmp`. Cannot be specified with `-Wsql`, `-Wsqlcomp`, `-Wmigrate`, or `-WIEEE_FLOAT`. You can use either or both *warn* or *verbose args*, separated by commas without space between them.

warn

Directs the SQL/MX compiler to return a warning rather than an error if a table does not exist at compile time.

verbose

Directs the SQL/MX compiler to display summary information, in addition to error and warning messages.

`-Wmxcmp_querydefault=compiler_attribute_name=compiler_attribute_value[,compiler_attribute_value...]`

Directs the SQL/MX compiler to issue the control query default setting at the command line. The command-line attribute settings override corresponding entries in the `SYSTEM_DEFAULTS` table. You can specify multiple attribute name and value pairs, separated by commas without spaces.

`-Wmxcmp_add=["args" | arg]`

Passes any valid set of `mxcmp` or `mxCompileUserModule` options (*args*) to the SQL/MX compiler (`mxcmp` or `mxCompileUserModule`) without change or validation. The SQL/MX compiler validates the syntax. You can specify multiple options and value pairs, separated by spaces.

`-Wmxcmp_files=args`

Passes the `.m` files specified here to `mxcmp` for module compilation (with module definition files). Passes all files without the `.m` file extension to `mxCompileUserModule` for module compilation (with embedded module definitions in the annotated source file).

<code>-WmoduleCatalog=arg</code>	Directs the SQL/MX preprocessor to use the catalog name if the input <i>sql-file</i> does not have a MODULE directive or its MODULE directive does not specify a catalog name.
<code>-WmoduleSchema=arg</code>	Directs the SQL/MX preprocessor to use the schema name if the input <i>sql-file</i> does not have a MODULE directive or its MODULE directive does not specify a schema name.
<code>-WmoduleGroup=[string]</code>	Directs the SQL/MX preprocessor to group all an application's module files. The <i>moduleGroup</i> is embedded in the module files' names and enables the use of OSS wild-card file specification patterns to manage the files. For more information, see Grouping on page 17-23.
<code>-WmoduleTableSet=[string]</code>	Directs the SQL/MX preprocessor to use the module management targeting feature. Create different sets of module files that can be used against different sets of tables. For more information, see Specifying the search locations of the module files on page 17-13.
<code>-WmoduleVersion=[string]</code>	Allows multiple versions of an application's module files to coexist while keeping the same MODULE directive in each version. For more information, see Versioning on page 17-21.
<code>-Wsqlmx_pp_defscript=args</code>	Specifies the files that contain the class MAP DEFINES that create environment variables before SQL/MX preprocessing.
<code>-Wmxcmp_cmd="oss_command; oss_command"</code>	Specifies the list of OSS commands to execute before invoking the remote <i>mxcmp</i> .
<code>-Wsqlhost={hostname IP-address}</code>	Specifies the host name or IP address of the NonStop system where the tables specified by INVOKE reside. This option is required if you use INVOKE. Note that if <code>-Wtarget=TNS/E</code> , the host must be an H-series (TNS/E) system.
<code>-Wsqlloc=OSS-directory</code>	Specifies the directory in which module definition files are placed.

<code>-Wsqlmx_port=port-number</code>	Specifies the TCP/IP port of the NonStop system to connect to for the ODBC listener process. The default port for the Association server is 18650.
<code>-Wsqluser=user[,password]</code>	Specifies the Guardian user name and password with access to the tables that INVOKE reads. Required if you use INVOKE.

The PC-only options are shaded in gray.

In addition to the options for preprocessing and compiling SQL/MX components, the `c89` utility supplies SQL/MX environment variables that provide the path name for the SQL/MX preprocessor (MXSQLC) and the SQL/MX compiler (MXCMP and MXCMPUM). For more information on `c89` environment variables, see the *Open System Services Shell and Utilities Reference Manual*.

SQL/MX Preprocessing

Use the `-Wsqlmx[=args]` command to invoke the SQL/MX preprocessor. When `-Wsqlmx` is specified, the `c89` utility also searches `/usr/tandem/sqlmx/include` for header files for the C/C++ compilers. For a full description of how the OSS-hosted SQL/MX preprocessor works, see [OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-19.

Compiling C/C++ Statements

Use the `c89` utility to compile the C/C++ statements in a preprocessed file to create an object (`.o`) file. The `c89` utility determines which compiler to use based on the file extension you use (`.ec` or `.sql` for embedded SQL/MX C programs, `.ecpp` for embedded SQL/MX C++ programs, and `.c` for embedded SQL/MP C programs). For a list of file extensions for SQL/MX source files, see [Program Files](#) on page 17-1.

Your application program can run in the OSS or Guardian environment. Use the `c89 -Wstype=oss` option (which is the default) if you want your application to be an OSS program. Use the `c89 -Wstype=guardian` option if you want your application to be a Guardian program. For more information, see [Building SQL/MX Guardian Applications in the OSS Environment](#) on page 15-72.

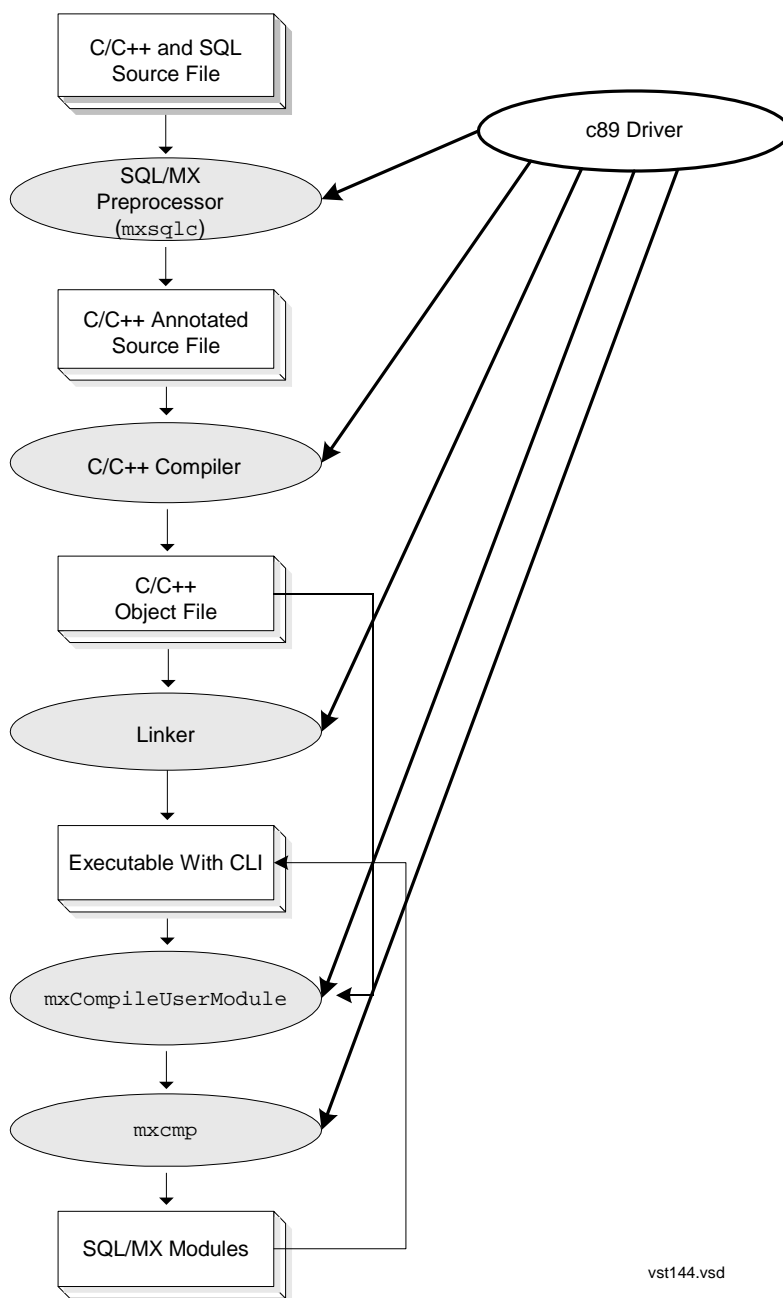
SQL/MX Compiling

Use the `-Wmxcmp[=args]` command to invoke the SQL/MX compiler. For a full description of how to use the SQL/MX compiler, see [Running the SQL/MX Compiler](#) on page 15-36.

c89 Examples With Embedded Module Definitions

Figure 15-3 shows how the `c89` utility compiles a C/C++ program with embedded module definitions.

Figure 15-3. c89 Generating Annotated Source With Embedded Module Definitions



vst144.vsd

- This command preprocesses, compiles, links, and SQL compiles a single C source file named `sqlprog.ec`:

```
c89 -Wsqlmx -Wmxcmp -o sqlprog.exe sqlprog.ec
```

The `c89` utility invokes the preprocessor, `mxsqlc`, which uses the file `sqlprog.ec` as input and produces one file, `sqlprog.c`, which is a C annotated source file that contains embedded module definitions. The `c89` utility then compiles and links `sqlprog.c` to produce the executable file, `sqlprog.exe`. The SQL/MX compiler command `-mxcmp` processes the executable file with the SQL/MX compiler, `mxCompileUserModule`, to produce the module.

- This command preprocesses several C++ source files and compiles them, but it does not link the results:

```
c89 -c -Wsqlmx file1.eC file2.ecc file3.ec++
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `file1.C`, `file2.cc`, `file3.c++`, `file1.o`, `file2.o`, and `file3.o`. TNS/E compilation also produces these files: `file1.dep`, `file2.dep`, and `file3.dep`.

- To compile the preprocessed source file named `sqlprog.cpp` without linking:

```
c89 -I /usr/tandem/sqlmx/include -c sqlprog.cpp \
-o sqlprog.exe
```

The `-I` option indicates the path for the `Platform.h` file.

- If you have multiple source files, you must compile each source file before linking them. For example, this `c89` command compiles the preprocessed source files named `sqlprog1.c` and `sqlprog2.c` and links them to create an executable file named `sqlprog.exe`:

```
c89 -I /usr/tandem/sqlmx/include sqlprog1.c sqlprog2.c \
-o sqlprog.exe
```

- The `c89` utility provides commands that pass through options to the SQL/MX preprocessor and the SQL/MX compiler (`-Wsqlmxadd` and `-Wmxcmp_add`, respectively). To preprocess files with preprocessor options, use the `-Wsqlmxadd` option:

```
-Wsqlmxadd=-a
```

To pass a single option, do not use quotes or white space characters. To pass multiple options, place them within double quotes and separate the options with a white-space character:

```
-Wsqlmxadd="-a -p -m -c test.c"
```

- If you did not link your object files using the `c89` utility, create the executable program by using the native object file linker to link one or more object files.

- For example, this `eld` command links `sqlprog1.o` and `sqlprog2.o` to create an executable file named `sqlprog.exe`:

```
eld /usr/lib/ccplmain.o sqlprog1.o sqlprog2.o \  
-o sqlprog.exe -lzcredll -lzcrtdll -lzosskdll \  
-lzicnv.dll -lzclidll
```

For more information on `eld`, see the *eld Manual*.

- For example, this `nld` command links `sqlprog1.o` and `sqlprog2.o` to create an executable file named `sqlprog.exe`:

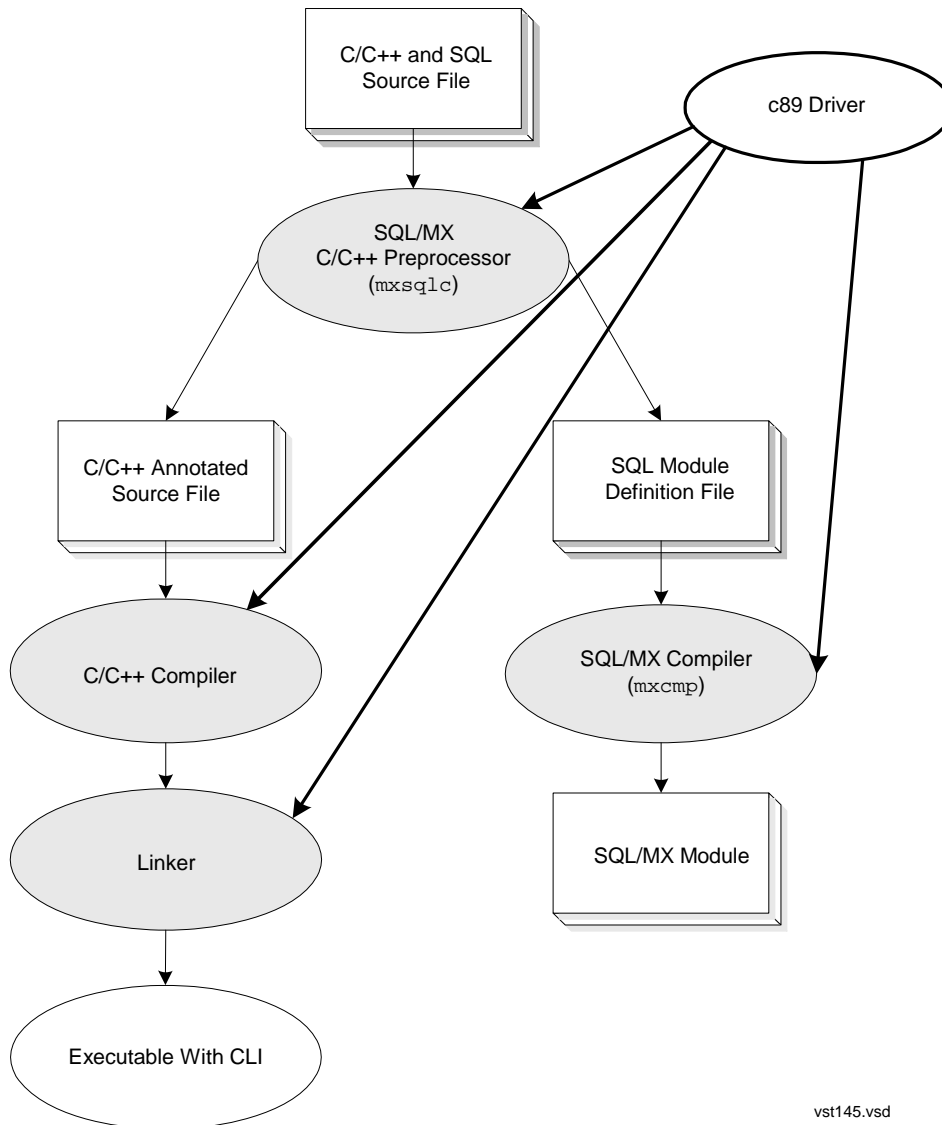
```
nld sqlprog1.o sqlprog2.o -o sqlprog.exe
```

For more information on `nld`, see the *nld Manual*.

c89 Examples With Module Definition Files

Figure 15-4 shows how the `c89` utility compiles a C/C++ program with module definition files.

Figure 15-4. c89 Generating Module Definition Files



vst145.vsd

- This command preprocesses, compiles, links, and SQL compiles a single C source file named `sqlprog.ec`:

```
c89 -Wsqlmx -Wsqlmxadd=-x -Wmxcmp -o sqlprog.exe \
sqlprog.ec sqlprog.m
```

The `c89` utility invokes the preprocessor, `mxsqlc`, which uses the file `sqlprog.ec` as input and produces two files: `sqlprog.c` and `sqlprog.m`. The file `sqlprog.c` is the C annotated source file, and the file `sqlprog.m` is the corresponding module definition file. The `c89` utility then compiles and links

`sqlprog.c` to produce the executable file, `sqlprog.exe`. The SQL/MX compiler command `-Wmxcmp` processes the module definition file `sqlprog.m` with the SQL/MX compiler, `mxcmp`, to produce the module.

- This command preprocesses several C++ source files and compiles them, but it does not link the results:

```
c89 -c -Wsqlmx -Wsqlmxadd=-x file1.eC file2.ecc file3.ec++
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `file1.m`, `file1.C`, `file2.m`, `file2.cc`, `file3.m`, `file3.c++`, `file1.o`, `file2.o`, and `file3.o`. TNS/E compilation also produces these files: `file1.dep`, `file2.dep`, and `file3.dep`.

- To compile the preprocessed source file named `sqlprog.cpp` without linking:

```
c89 -I /usr/tandem/sqlmx/include -c sqlprog.cpp -o \
sqlprog.exe
```

The `-I` option indicates the path for the `Platform.h` file.

- If you have multiple source files, you must compile each source file before linking them. For example, this `c89` command compiles the preprocessed source files named `sqlprog1.c` and `sqlprog2.c` and links them to create an executable file named `sqlprog.exe`:

```
c89 -I /usr/tandem/sqlmx/include sqlprog1.c sqlprog2.c \
-o sqlprog.exe
```

- If you did not link your object files by using the `c89` utility, create the executable program by using the native object file linker to link one or more object files.

- For example, this `eld` command links `sqlprog1.o` and `sqlprog2.o` to create an executable file named `sqlprog.exe`:

```
eld /usr/lib/ccplmain.o sqlprog1.o sqlprog2.o \
-o sqlprog.exe -lzcredll -lzcrtdll -lzoskdll \
-lzicnvdl1 -lzclidll
```

For more information on `eld`, see the *eld Manual*.

- For example, this `nld` command links `sqlprog1.o` and `sqlprog2.o` to create an executable file named `sqlprog.exe`:

```
nld sqlprog1.o sqlprog2.o -o sqlprog.exe
```

For more information on `nld`, see the *nld Manual*.

- The following command preprocesses, compiles, and then links a single SQL source file named `sqlprog.sql` in 64-bit mode:

```
c89 -Wcplusplus -Wlp64 -Wsqlmx -Wmxcmp -o sqlprog.exe
sqlprog.sql
```

- For 64-bit applications, if the object files are not linked using the `c89` utility, you must create the executable program using the native object file linker. You can link

one or more object files using the native object file linker. For example, the following `eld` command links `sqlprog1.o` and `sqlprog2.o` to create an executable file `sqlprog.exe`:

```
eld -set data_model lp64 /usr/lib/cmain64.o sqlprog1.o
sqlprog2.o -o sqlprog.exe -lycppcdll -lycpp3dll -lycredll -
lycrtldll -lyOSSKDLL -lyossfdll -lySECDLL -lyi18ndll -
lyicnvdll -lyOSSEDLL -lyINETDLL -lyOSSHDLL -lyOSSCDLL -
lyclidll
```

Examples of Building and Deploying Embedded SQL C/C++ Programs

The examples in this subsection use SQL/MP tables and SQL/MX Release 2.x.

Building a C/C++ Program With Embedded SQL Statements on Windows

You can build a C or C++ program by using the ETK product which is available on Microsoft Windows. Although the following example does not use ETK components, the ETK product is easy to use to create programs. For more information, see the ETK documentation.

This example illustrates how to build a C or C++ program on Windows that has two or more source files containing embedded SQL/MX statements that query SQL/MP tables into a self-contained application executable file. Suppose that your catalog is named FINANCE, and your schema is named WINDEV. Suppose that you want the location independence of class MAP DEFINES for your table names.

1. On your PC, create C or C++ source files (for example, `sqlprog1.sql` and `sqlprog2.sql`) that contain embedded SQL/MX statements:

```
// sqlprog1.sql
EXEC SQL MODULE sqlprog1mod; // externally qualified at SQL
preprocess-time
int main() {
EXEC SQL INVOKE =stocks AS stocks_type;
EXEC SQL DECLARE CURSOR s FOR SELECT * FROM =stocks;
...
}

// sqlprog2.sql
EXEC SQL MODULE sqlprog2mod; // externally qualified at SQL
preprocess-time
int prog2() {
EXEC SQL INVOKE =bonds AS bonds_type;
EXEC SQL DECLARE CURSOR b FOR SELECT * FROM =bonds;
...
}
```

2. Run the SQL/MX C/C++ preprocessor on each source file that has embedded SQL statements. Suppose that you want `mxsqlc` to expand `=stocks` to the SQL/MP table named `\pecan.$finance.assets.adrs` and `=bonds` to the SQL/MP table named `\pecan.$finance.assets.munis`. Suppose that you want `mxsqlc` to connect to the NonStop system `\pecan` under Guardian user `finance.tomr`, whose password is `abc123`. Suppose that you want `mxsqlc` to generate code that assumes the application's user modules will be in the schema named `'FINANCE.WINDEV'`. At a Windows command prompt, enter:

```
set stocks =\pecan.$finance.assets.adrs
set bonds =\pecan.$finance.assets.munis
```

```

mxsqlc sqlprog1.sql -c sqlprog1.cpp -s pecan \
-y finance.tomr -z abc123 \
-g moduleCatalog=FINANCE -g moduleSchema=WINDEV
mxsqlc sqlprog2.sql -c sqlprog2.cpp -s pecan \
-y finance.tomr -z abc123 \
-g moduleCatalog=FINANCE -g moduleSchema=WINDEV

```

These commands create two annotated source files (`sqlprog1.cpp` and `sqlprog2.cpp`) that contain the SQL/MX CLI call translations of the embedded SQL statements and extra C/C++ source constructs that represent the module definitions. These generated files will have hard-coded references to the modules named `FINANCE.WINDEV.sqlprog1mod` and `FINANCE.WINDEV.sqlprog2mod`. For security reasons, these module references cannot be remapped at run time.

3. Run the C/C++ compiler to compile the annotated source files into object files. Suppose that the SQL/MX CLI and other HP header files are in the `../include` directory.

```

c89 -Wversion2 -Wsqlmx -Wmxcmp -Wverbose -I/usr/include \
-c sqlprog1.ecpp sqlprog2.ecpp -o sqlprog

```

4. Transfer the object files (`sqlprog1.o` and `sqlprog2.o`) to the NonStop system (for example, `\pecan`).
5. Run the native linker to build a self-contained OSS executable file named `sqlprog`.

- For TNS/E native applications, use the `eld` utility:

```

eld -verbose /usr/lib/ccplmain.o sqlprog1.o sqlprog2.o \
-o sqlprog.exe -lzcpdcll -lzcpp2dll -lzcredll \
-lzcertdll -lzoskdll -lzi18ndll -lzicnvdll \
-lzclidll -lzt1h7dll

```

- For TNS/R native applications, use the `nld` utility:

```

nld -elf -set systype oss -set highpin off -set \
highrequestor on -set inspect on -obey \
/usr/lib/libc.obey -set saveabend on \
/usr/lib/crt1main.o sqlprog1.o sqlprog2.o \
-l zcplsr1 -l zcrtlsr1 -l zcresr1 -l zcplosr1 -l ztlhgsr1 \
-l ztlhosr1 -Bdynamic -l zclisr1 -o sqlprog

```

6. Set up needed `DEFINEs` and run the SQL/MX compiler:

```

add_define =stocks \
class=map file=\\pecan.\$finance.assets.adrs
add_define =bonds \
class=map file=\\pecan.\$finance.assets.munis
mxCompileUserModule -v -d AUTOMATIC_RECOMPILATION=ON \
-d RECOMPILATION_WARNINGS=ON -d SIMILARITY_CHECK=ON \
sqlprog

```

This command compiles the application's modules. The application (`sqlprog`) is now runnable on `\pecan`.

Developing a Native C/C++ Program With Embedded SQL/MX Statements on OSS

This example illustrates how to use OSS tools to build a C or C++ program that has two or more source files containing embedded SQL/MX statements into a self-contained application executable file. Suppose that your schema is named OSSDEV under the catalog named FINANCE.

1. Create C or C++ source files (for example, `sqlprog1.sql` and `sqlprog2.sql`) that contain embedded SQL/MX statements (and use SQL/MP tables):

```
// sqlprog1.sql
EXEC SQL MODULE sqlprog1mod; // externally qualified at SQL
preprocess-time
int main() {
EXEC SQL INVOKE =stocks AS stocks_type;
EXEC SQL DECLARE CURSOR s FOR SELECT * FROM =stocks;
...
}

// sqlprog2.sql
EXEC SQL MODULE sqlprog2mod; // externally qualified at SQL
preprocess-time
int prog2() {
EXEC SQL INVOKE =bonds AS bonds_type;
EXEC SQL DECLARE CURSOR b FOR SELECT * FROM =bonds;
...
}
```

2. Run the SQL/MX C/C++ preprocessor on each source file that has embedded SQL statements. Suppose that you want to place these application's user modules in the schema named 'FINANCE.OSSDEV' to distinguish your OSS-developed application modules from Windows-developed versions. As a result, you and the Windows-based developers can test your respective versions of `sqlprog` on the same NonStop system at the same time. Use class MAP DEFINES for `mssqlc` to correctly process the INVOKES of table DEFINES found in the sources.

```
add_define =stocks \
    class=map file=\\pecan.\$finance.assets.adrs
add_define =bonds \
    class=map file=\\pecan.\$finance.assets.munis
mssqlc sqlprog1.sql -c sqlprog1.cpp -g \
    moduleCatalog=FINANCE -g moduleSchema=OSSDEV
mssqlc sqlprog2.sql -c sqlprog2.cpp -g \
    moduleCatalog=FINANCE -g moduleSchema=OSSDEV
```

These commands create two annotated source files (`sqlprog1.cpp` and `sqlprog2.cpp`) that contain the SQL/MX CLI call translations of the embedded SQL statements and extra C/C++ source constructs that represent the module definitions. These files have hard-coded references to the modules named `FINANCE.OSSDEV.sqlprog1mod` and `FINANCE.OSSDEV.sqlprog2mod`. For security reasons, these module references cannot be remapped at run time.

3. Run the C/C++ compiler to compile the annotated source files into object files:

```
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c sqlprog1.cpp
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c sqlprog2.cpp
```

4. Run the native linker to build a self-contained OSS executable file named `sqlprog`.

- For TNS/E native applications, use the `eld` utility:

```
eld -verbose /usr/lib/ccplmain.o sqlprog1.o sqlprog2.o \
  sqlprog.exe -lzcpd11 -lzcpp2d11 -lzcred11 -lzctrl1d11 \
  -lzoskd11 -lzil8nd11 -lzicnv1d11 -lzclid11 -lztlh7d11
```

- For TNS/R native applications, use the `nld` utility:

```
nld -elf -set systype oss -set highpin off -set \
  highrequestor on -set inspect on -obey \
  /usr/lib/libc.obey -set saveabend on \
  /usr/lib/crt1main.o sqlprog1.o sqlprog2.o \
  -l zcplsr1 -l zcrtlsr1 -l zcresr1 -l zcplosr1 -l ztlhgsr1 \
  -l ztlhosr1 -Bdynamic -l zclisr1 -o sqlprog
```

5. Set up needed `DEFINES` and run the SQL/MX compiler:

```
add_define =stocks class=map \
  file=\\pecan.\$finance.assets.adrs
add_define =bonds class=map \
  file=\\pecan.\$finance.assets.munis
mxCompileUserModule -v -d AUTOMATIC_RECOMPILATION=ON \
  -d RECOMPILATION_WARNINGS=ON -d SIMILARITY_CHECK=ON sqlprog
```

This command compiles the application's module definitions. It does not overwrite the Windows-developed `sqlprog`'s modules because they use the module name prefix `'FINANCE.WINDEV'`. Assuming that the object `'sqlprog'` coming from Windows is at a different location than the one compiled and linked on OSS (for example, `/home/fin/windev/sqlprog` and `/home/fin/ossdev/sqlprog` respectively), you can now run, test, and debug both the Windows-developed `sqlprog` and the OSS-developed `sqlprog` simultaneously on `\pecan` without module confusion, interference, or accidental overwrites.

Building and Deploying a C Application With Embedded Module Definitions and Module Definition Files

Suppose that you have a set of SQL utility routines that were developed using SQL/MX Release 1.8. The object code is in `sqlutil.o`. Use these steps to build, statically link in `sqlutil.o`, and deploy a new application `sqlapp.exe` in the OSS environment.

1. Create C or C++ source files (for example, `sqlapp.sql`) that contain embedded SQL/MX statements:

```
// sqlapp.sql
EXEC SQL DECLARE SCHEMA 'cat.sch';
EXEC SQL MODULE sqlappmod; // might be externally qualified
      at SQL preprocess-time
```

```
EXEC SQL DECLARE CURSOR m FOR SELECT * FROM =midcaps;
int main() { ...
}
```

2. Run the SQL/MX C/C++ preprocessor:

```
mxsqlc sqlapp.sql -c sqlapp.cpp
```

This command creates an annotated source file (`sqlapp.cpp`) that contains the SQL/MX CLI call translations of the embedded SQL statements and an extra C/C++ source construct that represents its module definition.

Because no module specification strings were specified at the command line, `mxsqlc` generates code using the module name `'cat.sch.sqlappmod.'` The SQL/MX object naming rules for default catalog and schema apply to an unqualified module directive, in addition to other unqualified names of tables, views, and other SQL objects.

3. Run the C/C++ compiler on `sqlapp.cpp`:

```
c89 -Wversion2 -I/usr/tandem/sqlmx/include -c sqlapp.cpp
```

4. Run the native linker on `sqlapp.o` and `sqlutil.o` to create the `sqlapp.exe` executable file.

- For TNS/E native applications, use the `eld` utility:

```
eld -verbose /usr/lib/ccplmain.o sqlapp.o sqlutil.o
-o sqlapp.exe -lzcppedll -lzcpped2dll -lztlh7dll -lzcresdll \
-lzcrtdll -lzosskdll -lzi18ndll -lzicnv.dll -lzclidll
```

- For TNS/R native applications, use the `nld` utility:

```
nld -elf -set systype oss -set highpin off \
-set highrequestor on -set inspect on \
-obey /usr/lib/libc.obey -set saveabend on \
/usr/lib/crtlmain.o sqlapp.o sqlutil.o \
-lzcp1srl -lzcr1srl -lzcresrl -lzcpl1srl -lzt1hgsrl \
-lzt1hosrl -Bdynamic -lzclisrl -o sqlapp.exe
```

5. Run the SQL/MX compiler after setting up class MAP DEFINES:

```
add_define =midcaps class=map \
file=\\pecan.\$data07.holding.midcaps
mxCompileUserModule -v -d AUTOMATIC_RECOMPILATION=ON \
-d RECOMPILATION_WARNINGS=ON -d SIMILARITY_CHECK=ON \
sqlapp.exe
```

This command compiles the application's module `'cat.sch.sqlappmod.'` Assuming that the `sqlutil.o`'s compiled module is still current, you can now run `sqlapp.exe`.

Quick Builds and mxcmp Defaults in a One-File Application Deployment

Suppose that you are actively developing, testing, and debugging a new SQL/MX Release 2.x application that you have organized into three separate static SQL C/C++ source files. You can minimize unnecessary SQL recompilations during active development and still retain the simplicity of building and deploying self-contained application files by using the named module option of the SQL/MX compiler.

The next example uses `mxcmp` command-line defaults to module compile an application executable to work with one set of tables on the development system and later module compile the same application executable to work with a different set of tables on the production system.

In addition, this example shows that references to module names are always resolved early at preprocessing time. It also shows that the resolution of references to other SQL objects (tables, views, and so on) can be deferred to as late as module compilation time.

1. Create or edit static SQL C/C++ source files (for example, `s1.sql`, `s2.sql`, `s3.sql`) as needed:

```
// s1.sql
EXEC SQL MODULE s1m;
int sow(int a) { ...
EXEC SQL INSERT INTO seeds VALUES(?, ?, ?);
...
}

// s2.sql
EXEC SQL MODULE s2m;
int grow(int j) { ... EXEC SQL UPDATE trees SET h=h+? WHERE
id=?;
...
}

// s3.sql
EXEC SQL MODULE s3m;
int reap(int z) { ...
EXEC SQL DELETE crops WHERE id=?;
...
}
```

Notice that references to the SQL tables `seeds`, `trees`, and `crops` do not specify their catalog and schema so that the full resolution of these SQL object references can be deferred to as late as the module compilation step.

2. Run the SQL/MX C/C++ preprocessor to generate code that references modules named `cat.sch.s1m`, `cat.sch.s2m`, and `cat.sch.s3m`:

```
cd /usr/meas
mxsqlc s1.sql -g moduleSchema=cat.sch
mxsqlc s2.sql -g moduleSchema=cat.sch
mxsqlc s3.sql -g moduleSchema=cat.sch
```

These modules use the default `mxsqlc` option that generates the annotated output source files `s1.c`, `s2.c`, and `s3.c`.

3. Run the C/C++ compiler:

```
cd /usr/meas
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c s1.c
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c s2.c
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c s3.c
```

This step generates the object files `s1.o`, `s2.o`, and `s3.o`.

4. Run the native linker.

- For TNS/E native applications, use the `eld` utility:

```
cd /usr/meas
eld -verbose /usr/lib/ccplmain.o s1.o s2.o s3.o \
-o sprog.exe -lzcpdcdll -lzcpp2dll -lztlh7dll -lzcredll \
-lzctrlldll -lzosskdll -lzi18ndll -lzicnvdll -lzclidll
```

- For TNS/R native applications, use the `nld` utility:

```
cd /usr/meas
nld -elf -set systype oss -set highpin off \
-set highrequestor on -set inspect on \
-obey /usr/lib/libc.obey -set saveabend on \
/usr/lib/crtlmain.o s1.o s2.o s3.o \
-l zcplsr1 -l zctrlsr1 -l zcresr1 -l zcplosr1 -l ztlhgsr1 \
-l ztlhosr1 -Bdynamic -l zclisr1 -o sprog
```

This step generates the application executable object file `sprog`.

5. Set up needed class MAP DEFINES (none in this case), and run the SQL/MX compiler only on the module definitions that have not yet been compiled or that have changed recently:

```
mxCompileUserModule -e -v -d CATALOG=harvest -d \
  SCHEMA=second \
  /usr/meas/sprog {cat.sch.s1m,cat.sch.s2m,cat.sch.s3m}
```

This step generates plans that reference tables named `harvest.second.seeds`, `harvest.second.trees`, and `harvest.second.crops`.

6. Set up needed DEFINES (none in this case), and run and test the program.

```
sprog
```

7. Find the next bug and fix the offending source (for example, `s1.sql`), and repeat Step 1 to Step 6 for `s1.sql` only. Specifically, compile only the `cat.sch.s1m` module definition in Step 5:

```
mxCompileUserModule -e -v -d CATALOG=harvest -d \
  SCHEMA=second \
  /usr/meas/sprog {cat.sch.s1m}
```

- 8. Repeat Step 1 to Step 7 until the program is ready for deployment or until you cannot find any more bugs, whichever comes first. Transfer the program to its target deployment NonStop system (for example, \batman):

```
/usr/prod/sprog.
```

- 9. Set up needed DEFINES (none in this case), and run the SQL/MX compiler on the application executable:

```
mxCompileUserModule -e -v -d CATALOG=bread -d \
    SCHEMA=basket /usr/prod/sprog
```

This step generates plans that reference tables named bread.basket.seeds, bread.basket.trees, and bread.basket.crops in the modules named cat.sch.s1m, cat.sch.s2m and cat.sch.s3m.

- 10. Set up needed DEFINES (none in this case), and run and test the program:

```
/usr/prod/sprog
```

Deploying a Static SQL Application to an RDF System

This example shows a method to develop and deploy a static SQL C/C++ application in an SQL/MX Release 2.x RDF system. In SQL/MX Release 2.x, the RDF primary system's SQL/MX catalog and schema names might be different from the corresponding SQL/MX catalog and schema names on the backup system.

Suppose that you want to do all development and module compilations only on the development system, and you do not want to do module compilations on the RDF systems if possible. Suppose that you have all your data in SQL/MP tables. To get data location independence, suppose that you prefer to use class MAP DEFINES instead of PROTOTYPE host variables for your application's table names. [Table 15-2](#) lists the module schemas and export files used in the following discussion.

Table 15-2. Module Schemas and Export Files for RDF SQL Application Deployment Example (page 1 of 2)

	Development	RDF Primary	RDF Backup
Node name	\ROBIN	\APPLE	\INDUS
Module schema for primary	TELCO.MODULES	TELCO.MODULES	
Module schema for backup	COMMS.MODS		COMMS.MODS
OSS directory for primary	/usr/primary	/usr/alpha	

**Table 15-2. Module Schemas and Export Files for RDF SQL Application
Deployment Example** (page 2 of 2)

	Development	RDF Primary	RDF Backup
OSS directory for backup	/usr/backup		/usr/beta
=debits DEFINE for primary	\$data07.ccards.debits	\$plat.charges.buys	
=debits DEFINE for backup	\$data07.ccards.debits		\$gold.cards.debits
=credits DEFINE for primary	\$data17.ccards.credits	\$plat.charges.pays	
=credits DEFINE for backup	\$data17.ccards.credits		\$gold.cards.credit

1. Suppose that the RDF primary system is \APPLE. On \APPLE, suppose that you want all your user modules to use the schema 'telco.modules'.
2. Suppose that the RDF backup system is \indus. On \indus, suppose that you are constrained by SQL/MX Release 2.x RDF rules to use the schema 'comms.mods'.
3. On the development system \robin, create a development (possibly, zero-row or empty) copy of your application's SQL/MP tables (and their statistics) from the RDF primary system \APPLE. You need not make local copies of the backup system's corresponding tables because they will always be similar to the primary system's tables.
4. On the development system \robin, create static SQL C/C++ source files (for example, nonstop1.sql and nonstop2.sql):

```
// nonstop1.sql
EXEC SQL MODULE nonstop1mod;
int main() { ...
EXEC SQL INVOKE =debits AS debits_type;
EXEC SQL UPDATE =debits SET balance = balance + ?;
...
}

// nonstop2.sql
EXEC SQL MODULE nonstop2mod;
int pay() { ...
EXEC SQL INVOKE =credits AS credits_type;
EXEC SQL UPDATE =credits SET balance = balance + ?;
...
}
```

5. On the development system \robin, set up a separate directory for building the RDF backup \indus version of the self-contained application executable. Set up DEFINES, run the preprocessor, the C/C++ compiler, native linker, and SQL/MX compiler.

- For TNS/E native applications:

```
cd /usr/backup
add_define =debits class=map \
    file=\\robin.$data07.ccards.debits
add_define =credits class=map \
    file=\\robin.$data17.ccards.credits
mxsqlc nonstop1.sql -c nonstop1.cpp -g \
    moduleSchema=comms.mods
mxsqlc nonstop2.sql -c nonstop2.cpp -g \
    moduleSchema=comms.mods
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c nonstop1.cpp
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c nonstop2.cpp
eld -verbose /usr/lib/cpclmain.o nonstop1.o nonstop2.o \
-o nonstop.exe -lzcppcdll -lzcpp2dll -lzt1h7dll -lzcredll \
-lzcrtdll -lzoskdll -lzi18ndll -lzicnvdl1 -lzclidll

mxCompileUserModule -d SIMILARITY_CHECK=on -d \
    AUTOMATIC_RECOMPILATION=on nonstop.exe
```

- For TNS/R native applications:

```
cd /usr/backup
add_define =debits class=map \
    file=\\robin.$data07.ccards.debits
add_define =credits class=map \
    file=\\robin.$data17.ccards.credits
mxsqlc nonstop1.sql -c nonstop1.cpp -g \
    moduleSchema=comms.mods
mxsqlc nonstop2.sql -c nonstop2.cpp -g \
    moduleSchema=comms.mods
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c nonstop1.cpp
c89 -Wversion2 -I /usr/tandem/sqlmx/include -c nonstop2.cpp
nld -elf -set systype oss -set highpin off \
    -set highrequestor on -set inspect on \
    -obey /usr/lib/libc.obey -set saveabend on \
    /usr/lib/crt1main.o nonstop1.o nonstop2.o -l zcplsr1 \
    -l zcrtlsr1 -l zcresr1 -l zcplosr1 -l ztlhgsr1 \
    -l ztlhosr1 -Bdynamic -l zclisr1 -o nonstop.exe

mxCompileUserModule -d SIMILARITY_CHECK=on -d \
    AUTOMATIC_RECOMPILATION=on nonstop.exe
```

6. On the development system \robin, set up a separate directory for building the RDF primary \APPLE version of the self-contained application executable. Set up DEFINES and run the preprocessor, the C/C++ compiler, native linker, and SQL/MX compiler.

- For TNS/E native applications:

```
cd /usr/primary
add_define =debits class=map \
    file=\\robin.$data07.ccards.debits
add_define =credits class=map \
    file=\\robin.$data17.ccards.credits
mxsqlc nonstop1.sql -c nonstop1.cpp -g \
    moduleSchema=telco.modules
mxsqlc nonstop2.sql -c nonstop2.cpp -g \
    moduleSchema=telco.modules
c89 -I /usr/tandem/sqlmx/include -c nonstop1.cpp
c89 -I /usr/tandem/sqlmx/include -c nonstop2.cpp
eld -verbose /usr/lib/ccplmain.o nonstop1.o nonstop2.o
-o nonstop.exe -lzcpcdll -lzcpc2dll -lzt1h7dll \
-lzcredll -lzcrtdll -lzoskdll -lzi18ndll \
-lzicnvdll -lzclidll

mxCompileUserModule -d SIMILARITY_CHECK=on -d \
    AUTOMATIC_RECOMPILATION=on nonstop.exe
```

- For TNS/R native applications:

```
cd /usr/primary
add_define =debits class=map \
    file=\\robin.$data07.ccards.debits
add_define =credits class=map \
    file=\\robin.$data17.ccards.credits
mxsqlc nonstop1.sql -c nonstop1.cpp -g \
    moduleSchema=telco.modules
mxsqlc nonstop2.sql -c nonstop2.cpp -g \
    moduleSchema=telco.modules
c89 -I /usr/tandem/sqlmx/include -c nonstop1.cpp
c89 -I /usr/tandem/sqlmx/include -c nonstop2.cpp
nld -elf -set systype oss -set highpin off \
    -set highrequestor on -set inspect on -obey \
    /usr/lib/libc.obey -set saveabend on \
    /usr/lib/crt1main.o nonstop1.o nonstop2.o \
    -l zcplsr1 -l zcrtlsr1 -l zcresr1 -l zcplosr1 -l ztlhgsr1 \
    -l ztlhosr1 -Bdynamic -l zclisr1 -o nonstop.exe

mxCompileUserModule -d SIMILARITY_CHECK=on -d \
    AUTOMATIC_RECOMPILATION=on nonstop.exe
```

7. Run and test both versions of the application nonstop.exe.
8. Display and verify the compiled plans of the user modules for the RDF primary version:

```
mxci
select * from table(explain \
    ('telco.modules.nonstop1mod', '%'));
select * from table(explain \
    ('telco.modules.nonstop2mod', '%'));
```

9. Display and verify the compiled plans of the user modules for the RDF backup version:

```
mxci
select * from table(explain('comms.mods.nonstop1mod', '%'));
select * from table(explain('comms.mods.nonstop2mod', '%'));
```

10. From the development system \robin, transfer the application executable nonstop.exe to the RDF backup system \indus (for example, into the /usr/beta directory).
11. As super ID, transfer the application's modules (comms.mods.nonstop1mod, comms.mods.nonstop2mod) from the development system to the /usr/tandem/sqlmx/USERMODULES directory of the RDF backup system.
12. On the RDF backup system \indus, log on as the application operator, set up DEFINES, and get ready to take over and run the application nonstop.exe if the RDF primary system fails:

```
add_define =debits class=map file=\\indus.$gold.cards.debits
add_define =credits class=map \
  file=\\indus.$gold.cards.credits
-- now ready to take over and run nonstop.exe
```

13. From the development system \robin, transfer the application executable nonstop.exe to the RDF primary system \APPLE (for example, into the /usr/alpha directory).
14. As super ID, transfer the application's modules (telco.modules.nonstop1mod, telco.modules.nonstop2mod) from the development system to the /usr/tandem/sqlmx/USERMODULES directory of the RDF primary system.
15. On the RDF primary system \APPLE, log on as the application operator, set up DEFINES, and run the application:

```
add_define =debits class=map file=\\APPLE.$plat.charges.buys
add_define =credits class=map \
  file=\\APPLE.$plat.charges.pays nonstop.exe
```

Building SQL/MX C/C++ Applications to Run in the Guardian Environment

To build SQL/MX C/C++ applications that run in the Guardian environment, choose one of these approaches, depending on your preferred development environment:

- [Building SQL/MX Guardian Applications in the Guardian Environment](#) on page 15-67
- [Building SQL/MX Guardian Applications in the OSS Environment](#) on page 15-72

Building SQL/MX Guardian Applications in the Guardian Environment

- [Using the OSS Pass-Through Command](#) on page 15-67
- [OSS-to-Guardian File Naming](#) on page 15-67
- [Steps for Building an SQL/MX C Application in the Guardian Environment](#) on page 15-68
- [Using a TACL Macro to Build a C Guardian Application](#) on page 15-69
- [Steps for Building an SQL/MX C++ Application in the Guardian Environment](#) on page 15-70
- [Using a TACL Macro to Build a C++ Guardian Application](#) on page 15-71

Using the OSS Pass-Through Command

Most commands for building an SQL/MX Guardian application can be issued directly at a TACL prompt. However, the SQL/MX preprocessor, `mxsqlc`, and the SQL/MX compiler, `mxcmp`, are OSS commands, which run only from OSS. Although `mxcmp` resides in the Guardian environment, it runs as an OSS process and must be started in the OSS environment.

To run the SQL/MX preprocessor and the SQL/MX compiler in the Guardian environment, use the OSS pass-through command by specifying the `osh -c` option. The `osh -c` option executes one command line at a time in the OSS environment. When you use the `osh -c` command, remember to enclose the entire command string after `osh -c` in double quotes.

Note. When using OSS pass-through commands in the Guardian environment, be aware of the effect of #INFORMAT TACL on those commands. If #INFORMAT TACL is in effect for your session, you must put a tilde (~) before the pipe (|) symbol. Otherwise, the pipe symbol cannot reach the shell for execution because it has a programming function within TACL.

OSS-to-Guardian File Naming

When you issue OSS commands from a TACL prompt to preprocess and SQL compile an application, the Guardian file names change automatically. In the Guardian environment, the period is automatically dropped from the file name.

For example, this OSS pass-through command preprocesses a C source file and generates an annotated source file and module definition file in the Guardian environment:

```
TACL> osh -c "mxsqlc prog.sql -c prog.c -m prog.m"
```

The annotated source file and module definition file in \$MYVOL.MYSUBVOL are `PROGC` and `PROGM`. Be aware of the Guardian file name limitation of eight characters.

Steps for Building an SQL/MX C Application in the Guardian Environment

Use the next commands at a TACL prompt to preprocess, SQL compile, and compile and link an SQL/MX C program.

1. To make the source file in the Guardian environment accessible to an OSS process, enter this command, replacing *myvol.mysubvol* with your default Guardian volume and subvolume:

```
param home /G/myvol/mysubvol
```

The source file named *progsq1* in *\$MYVOL.MYSUBVOL* must be Guardian file code 101.

2. To invoke the SQL/MX preprocessor, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "mxsqlc progsq1 -c progC -m progM ~|tee templog"
```

3. To invoke the SQL/MX compiler, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "/G/system/system/mxcmp progM ~|tee -a templog"
```

4. Errors generated by the SQL/MX preprocessor or SQL/MX compiler are logged in the OSS file *templog*. To convert the error log to a Guardian file:

```
TACL> purge proglog
TACL> ctoedit templog,proglog
```

5. Run the Guardian C compiler and linker.

For TNS/E native compilation:

```
== Convert the annotated source file from an OSS text file
== (file code 180) to a Guardian text file (file code 101).
TACL> ctoedit progC,progsrC
== Call the CCOMP compiler to generate the object file.
TACL> ccomp/in progsrC,out progout/progo
== Call the eld linker to generate an executable file.
TACL> eld $system.system.ccplmain progO -o progexe &
        -lzcredll -lzcrtdll -lzosskdll -lzi18ndll &
        -lzicnvdl1 -lzclidll
```

For TNS/R native compilation:

```
== Convert the annotated source file from an OSS text file to
== a Guardian text file.
TACL> ctoedit progC,progsrC
== Call the NMC compiler to generate the object file.
TACL> nmc/in progsrC,out progout/progo
== Call the nld linker to generate an executable file.
TACL> nld $system.system.crtlmain progO -o progexe &
        -obey $system.system.libcobey -lzclisrl -lzcplsr1 &
        -lzt1hsrl -noverbose
```

6. Execute the executable:

```
TACL> run progexe
```

Using a TACL Macro to Build a C Guardian Application

Use a TACL macro file to combine and execute the commands. Use these sample TACL macros to customize your own script. In the samples, the source file is located in the Guardian environment and named `progsq1`. Remember that the source file must be Guardian file code 101.

For TNS/E native compilation:

```
?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
osh -c "mxsqlc progsq1 -c progC -m progM ~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp progM ~|tee -a templog"
== Convert OSS text files (file code 180) to Guardian text files
== (file code 101).
sink [#purge proglog]
ctoedit templog,proglog
ctoedit progC,progsrc
== Call the CCOMP compiler to generate the object file.
ccomp/in progsrc,out progout/progo;nowarn
== Call the eld linker to generate an executable file.
eld $system.system.ccplmain progO -o progexe &
-lzcredll -lzcrtdll -lzosskdll -lzi18ndll -lzicnvdl1 &
-lzclidl1
== Execute the executable.
run progexe
```

For TNS/R native compilation:

```
?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
osh -c "mxsqlc progsq1 -c progC -m progM ~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp progM ~|tee -a templog"
== Convert OSS text files to Guardian text files.
sink [#purge proglog]
ctoedit templog,proglog
ctoedit progC,progsrc
== Call the NMC compiler to generate the object file.
nmc/in progsrc,out progout/progo;nowarn
== Call the nld linker to generate an executable file.
nld $system.system.crtlmain progO -o progexe -obey &
$system.system.libcobey -lzclisrl -lzcplsr1 -lztlhrsrl -noverbose
```

```
== Execute the executable.
run progexe
```

Steps for Building an SQL/MX C++ Application in the Guardian Environment

Use the next commands at a TACL prompt to preprocess, SQL compile, and compile and link an SQL/MX C++ program.

1. To make the source file in the Guardian environment accessible to an OSS process, enter this command, replacing *myvol.mysubvol* with your default Guardian volume and subvolume:

```
param home /G/myvol/mysubvol
```

The source file named *progcpp* in *\$MYVOL.MYSUBVOL* must be Guardian file code 101.

2. To invoke the SQL/MX preprocessor, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "mxsqlc progcpp -c progcpp -m prog &
|tee templog"
```

3. To invoke the SQL/MX compiler, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "/G/system/system/mxcmp prog | tee -a templog"
```

4. Errors generated by the SQL/MX preprocessor or SQL/MX compiler are logged in the OSS file *templog*. To convert the error log to a Guardian file:

```
TACL> purge proglog
TACL> ctoedit templog,proglog
```

5. Run the Guardian C++ compiler and linker.

For TNS/E native compilation, you can use either the default C++ library, which is *version3*, or *version2* in your CPPCOMP compiler command:

```
== Convert the annotated source file from an OSS text file
== (file code 180) to a Guardian text file (file code 101).
TACL> ctoedit progcpp,progrsrc
== Call the CPPCOMP compiler to generate the object file.
TACL> cppcomp/in progrsrc,out progout/progo;version2
== Call the eld linker to generate an executable file.
TACL> eld $system.system.ccplmain progo -o progexe &
        -lzcpcdll -lzcpc2dll -lzcrcdll -lzcrtldll &
        -lzsskdll -lzi18ndll -lzicnvdl1 -lzcldll -lzt1h7dll
```

For TNS/R native compilation, you can use either the default C++ library, which is *version3*, or *version2* in your NMCPLUS compiler command:

```
== Convert the annotated source file from an OSS text file to
== a Guardian text file.
TACL> ctoedit progcpp,progrsrc
```

```

== Call the NMCPLUS compiler to generate the object file.
TACL> nmcplus/in progsrcl,out progout/progo;version2
== Call the nld linker to generate an executable file.
TACL> nld $system.system.crtlmain prog -o progexe -obey &
        $system.system.libcobey -lzclisrl -lzcplsrsl &
        -lztlhsrl -noverbose

```

6. Execute the executable:

```
TACL> run progexe
```

Using a TACL Macro to Build a C++ Guardian Application

Use a TACL macro file to combine and execute the commands. Use these sample TACL macros to customize your own script. In the samples, the source file is located in the Guardian environment and named `progecpp`. Remember that the source file must be Guardian file code 101.

For TNS/E native compilation:

```

?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
osh -c "mxsqlc progecpp -c progcpp -m prog ~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp prog ~|tee -a templog"
== Convert OSS text files (file code 180) to Guardian text files
== (file code 101).
sink [#purge proglog]
ctoedit templog,proglog
ctoedit progcpp,progsrcl
== Call the CPPCOMP compiler to generate the object file.
cppcomp/in progsrcl,out progout/progo;version2
== Call the eld linker to generate an executable file.
eld $system.system.ccplmain prog -o progexe &
-lzcpcddl -lzcpc2ddl -lzcrcddl -lzcrtldll -lzosskddl &
-lzi18ndll -lzi18ndll -lzi18ndll -lzi18ndll
== Execute the executable.
run progexe

```

For TNS/R native compilation:

```

?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
osh -c "mxsqlc progecpp -c progcpp -m prog ~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp prog ~|tee -a templog"
== Convert OSS text files to Guardian text files.
sink [#purge proglog]
ctoedit templog,proglog

```

```
ctoedit progcpp,progrsrc
== Call the NMCPLUS compiler to generate the object file.
nmcplus/in progrsrc,out progout/progo;version2
== Call the nld linker to generate an executable file.
nld $system.system.crtlmain progout -o progexe -obey &
$system.system.libcobey &
-lzclisrl -lzcplisrl -lztlsrl -noverbose
== Execute the executable.
run progexe
```

Building SQL/MX Guardian Applications in the OSS Environment

You can use the `c89 -Wsysstype=guardian` option to build an SQL/MX Guardian application in the OSS environment. Follow these steps:

1. Create an embedded SQL/MX C/C++ source file (for example, `prog.ec`) in the OSS environment.
2. Compile the C/C++ source file by using the `-Wsysstype=guardian` option of the OSS compiler utility:

```
c89 -Wsqlmx -Wmxcmp -Wsysstype=guardian prog.ec \
-o prog.exe
```

3. Copy the executable file, `prog.exe`, from an OSS directory to a Guardian volume and subvolume:

```
cp prog.exe /G/myvol/mysubvol/progexe
```

4. In the Guardian environment, assign file code 800 (for TNS/E native applications) or file code 700 (for TNS/R native applications) to the executable file:

```
TACL> fup alter progexe, code 800
TACL> fup alter progexe, code 700
```

5. Run the executable in the Guardian environment:

```
TACL> run progexe
```

Running an SQL/MX Application

This subsection describes how C or C++ application code is correctly linked to the compiled SQL/MX user module. Topics include:

- [Running the SQL/MX Program File](#) on page 15-73
- [Understanding and Avoiding Some Common Run-Time Errors](#) on page 15-73
- [Debugging a Program](#) on page 15-75
- [Displaying Query Execution Plans](#) on page 15-75

As stated in [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8, when the preprocessor reads an embedded SQL source file and writes the C/C++ annotated source file, it replaces the SQL statements with C code to call the SQL CLI to execute

the SQL statement, along with code to handle parameter passing and error processing. At run time, the calls to the CLI pass in a descriptor of the statement, which gives the statement name, the module name, and a module timestamp.

The CLI begins processing each call by checking that it has the associated module in memory. If not, it uses the module name to find the correct module file in the application's base directory. If a co-located module is not found there, it looks for the module file in the `/usr/tandem/sqlmx/USERMODULES` directory. Before it reads in the compiled SQL plans from a module file, the CLI also checks that the module timestamp encoded in the module file matches the module timestamp passed in from the C/C++ application.

If the application consists of more than one separately compiled module, when the first statement from the module is executed, the sequence of reading the module file and checking its module timestamp is performed and repeated for each module associated with the application.

Security of the `/usr/tandem/sqlmx/USERMODULES` directory is very important. You should restrict access so that users cannot alter the query plans in the modules or remove modules. For information on securing modules, see the *SQL/MX Release 3.2 Management Manual*.

Running the SQL/MX Program File

An SQL/MX program can run in the OSS or in the Guardian environment. You can use the `GTACL` command to start a Guardian program from OSS. You can use the `osh` command to start an OSS program from a Guardian TACL session.

- From the OSS environment, enter the program file name at the OSS shell prompt. You can also use the OSS `run` command to run the program file by using specific Guardian attributes (for example, a CPU or priority for the process).
- From the Guardian environment, use the TACL `osh` command to run the program. For more information, see [Building SQL/MX C/C++ Applications to Run in the Guardian Environment](#) on page 15-66.

For more information on the `run` or `osh` command, see the *Open System Services Shell and Utilities Reference Manual* or the OSS reference pages.

Understanding and Avoiding Some Common Run-Time Errors

The details of how a C/C++ executable is linked with its module or modules are handled by the system and take place in the background. However, by understanding this process and why certain run-time errors occur, you can avoid some common SQL/MX application development issues.

Module File Errors

Error 8809 Unable to open module file

Error 8809 error occurs if module files are deleted from the base directory of the application, the `/usr/tandem/sqlmx/USERMODULES` directory, user-specified Guardian or OSS location(s) or both, or the application DLL location(s).

This error might also occur if the named module file exists but is not readable, or if the required permission to access the volume, sub volume, or the OSS directory is not granted. The owner of the module file must change the permission attributes to ensure that an application can read the module file.

Error 8808 Module file contains corrupted or invalid data

This error occurs when the timestamp encoded in the module file does not match the timestamp passed from the application to the CLI. These timestamps are initially generated by the preprocessor and are used to ensure that the version of the application is synchronized with the version of the module file. This error can occur if you run the preprocessor on your embedded SQL, compile the annotated C/C++ output file, but fail to SQL compile the module definition file that the preprocessor generates. If the SQL/MX compiler has previously compiled a different instance of the module definition file, a module exists whose name corresponds to the application module but has a mismatched timestamp.

This error can also occur if you make a copy of an application executable, rebuild the application (thus overwriting the original instance of the application's module file), and then execute the first copy of the application.

A common cause of error 8808 is reuse of code. If you have an embedded SQL source named `myutils.sql`, you might build and link `myutils` with a number of applications. Each build (that is, preprocessing, c89-compilation, and SQL compilation) of `myutils` results in a new copy of the same module file overwriting an earlier copy. Only the last application built with `myutils.sql` avoids error 8808.

To avoid error 8808:

- If you want to reuse embedded modules, use either the grouping or versioning attributes described in [Section 17, Program and Module Management](#). Qualifying your module name with a group or version attribute enables the separate builds of a module to coexist.
- Build `myutils.sql` only once, and then link the resulting `myutils.o` file to each application.

When you need to rebuild `myutils` for each application, you can either edit the `myutils.sql` source and change the name of the module that you give in the `MODULE` directive, or you can avoid the `MODULE` directive and let the preprocessor generate the module name.

Error 8400 The CLASS attribute of the DEFINE is not correct.

Error 8400 occurs if the `Define =_MX_MODULE_SEARCH_PATH` CLASS type is not SEARCH DEFINE. This variable is used to locate and load the module file. Ensure that `Define =_MX_MODULE_SEARCH_PATH` is specified correctly and restart the embedded SQL program.

Module File Naming

In application development, avoid the use of delimited identifiers that contain dots (.) in the name of a module's catalog and schema and in the module name itself. Delimited identifiers begin and end with double quotation characters (" "). However, quotation characters are removed when NonStop SQL/MX forms the three-part module name. In some cases of delimited identifiers that contain dots, the resulting three-part module name duplicates an unrelated module name, replacing the query execution plans of the other module file. For example, a module named "A.B".C.D (catalog "A.B", schema C, and module name D) creates a module file name of `/usr/tandem/sqlmx/USERMODULES/A.B.C.D`. A module named A."B.C".D (catalog A, schema "B.C", and module name D) creates an identically named module file. The second file overwrites the first, and the first module's application cannot execute. For more information on delimited identifiers, see the *SQL/MX Reference Manual*.

Debugging a Program

You can debug a C/C++ program and its corresponding SQL/MX module by using:

- **Native Inspect.** A system-level command-line symbolic debugger that can be used to debug TNS/E native programs. For more information, see the *Native Inspect Manual*.
- **Inspect:** A symbolic interactive debugger that provides both machine-level and source-level debugging for TNS/R native programs. To run Inspect in the OSS environment, enter this command at the OSS prompt:

```
run -debug -inspect=on sqlprog.exe
```

where `sqlprog.exe` is the SQL/MX program you are debugging.

For detailed information, see the *Inspect Manual*.

- **Visual Inspect:** A symbolic debugger that provides source-level debugging with a graphical user interface (GUI). Visual Inspect is a client-server application. The server component runs on an HP NonStop operating system, and the client component runs on a workstation in the Windows environment. Detailed documentation is available in the client component online help.

Displaying Query Execution Plans

The EXPLAIN function is an SQL/MX extension that generates a result table describing an access plan for a DML statement, otherwise known as a *query execution plan*. Use

the EXPLAIN function for a DML statement in a module. For more information on the EXPLAIN function, see the *SQL/MX Reference Manual* and the *SQL/MX Query Guide*.

Note. If there is no EXPLAIN output for a statically compiled application, the GENERATE_EXPLAIN default attribute might have been turned off during compilation. In this case, verify that GENERATE_EXPLAIN is on and recompile the application.

Displaying the Query Execution Plan of One Statement

To display the EXPLAIN output for a specific DML statement in a module, issue this statement in MXCI:

```
SELECT * FROM TABLE(EXPLAIN( 'module-name' ,
                               'statement-pattern' ));
```

Module Name

The *module-name* is the full name of a module, is case-sensitive, and must be placed within single quotes:

```
'CAT.SCH.GRP1^MOD1^TABLESET1^VER1'
```

The module name is either specified by the MODULE directive in the embedded SQL source file or by the preprocessor-generated module name if you did not use a MODULE directive. For more information on the module name, see [Module Management Naming](#) on page 17-9.

Note. Do not confuse module files, which do not have file extensions and reside in the application's base directory or in the `/usr/tandem/sqlmx/USERMODULES` directory, with module definition files (`.m`), which are optionally generated during preprocessing and are precursors to modules. For more information, see [Module Management Naming](#) on page 17-9.

Statement Pattern

The *statement-pattern* is the name of a DML statement in the module. The *statement-pattern* is case-sensitive and must be placed within single quotes:

```
'MX_DEFAULT_STATEMENT_0'
```

To determine the name of a particular SQL statement, if you SQL compiled your module with:

- `mxcmp`, look in the module definition file. The module definition file is an ASCII file that you can view.
- `mxCompileUserModule` (creating an annotated source file with embedded module definitions), you cannot simply view the annotated source file as it is a binary file. You can determine the statement names if you know the module name. For more information on the module name, see [Module Name](#) on page 15-76.

You can use the following query from MXCI to determine the module's statement names and associated SQL queries, substituting the actual value of your module name in place of 'CAT.SCH.MYMOD'.

```
select module_name, statement_name,
       cast(trim(substring(description from
       (position('statement_index: ' in description) + 17)
       for (position(' ' in substring(
       description from
       (position('statement_index: ' in description) + 17))))))
       as integer) as stmt_index,
       substring(description from position(
       statement: ' in description) + 11 for 9999) as stmt
from table(explain('CAT.SCH.MYMOD', '%'))
where operator = 'ROOT'
order by stmt_index;
```

The query displays output similar to:

MODULE_NAME	STATEMENT_NAME	STMT_INDEX	STMT
CAT.SCH.MYMOD	SQLMX_DEFAULT_STATEMENT_1	0	PROCEDURE
C1 () INSERT INTO T VALUES(1);			
CAT.SCH.MYMOD	SQLMX_DEFAULT_STATEMENT_2	1	PROCEDURE
SQL_DEFAULT_STATEMENT_1() COMMIT WORK			
---2 row(s) selected.			

Displaying the Query Execution Plan of All Statements

To display the EXPLAIN output for all DML statements in a module, issue this statement in MXCI:

```
SELECT * FROM TABLE(EXPLAIN('module-name', '%'));
```

Module Name

The *module-name* is the full name of a module, is case-sensitive, and must be placed within single quotes:

```
'CAT.SCH.GRP1^MOD1^TABLESET1^VER1'
```

The module-name is either specified by the MODULE directive in the embedded SQL program or the preprocessor generated module name if you did not use a MODULE directive. For more information on the module name, see [Module Management Naming](#) on page 17-9.

Note. Do not confuse module files, which do not have file extensions and reside in the application's base directory or in the /usr/tandem/sqlmx/USERMODULES directory, with module definition files (.m), which are optionally generated during preprocessing and are precursors to modules. For more information, see [Module Management Naming](#) on page 17-9.

Wild Card (%)

Instead of specifying a statement pattern, use the percent sign (%) to represent all the DML statements in the module. The percent sign (%) must be placed within single quotes:

' % '

For information on how to interpret the output of the EXPLAIN function, see the *SQL/MX Query Guide*.

16 COBOL Program Compilation

This section describes how to develop and execute a COBOL program that contains embedded SQL statements. In addition, this section contains information on embedded module definitions and module definition files:

- [Compiling SQL/MX Applications and Modules](#) on page 16-2
- [Running the SQL/MX COBOL Preprocessor](#) on page 16-9
- [Running the COBOL Compiler and Linker](#) on page 16-23
- [Running the SQL/MX Compiler](#) on page 16-25
- [ecobol or nmcobol Utility: Using One Command for All Compilation Steps](#) on page 16-33
- [Combining Embedded Module Definitions and Module Definition Files](#) on page 16-46
- [Building SQL/MX COBOL Applications to Run in the Guardian Environment](#) on page 16-47
- [Running an SQL/MX Application](#) on page 16-51

For information on managing COBOL programs and SQL/MX modules, see [Section 17, Program and Module Management](#).

Compiling SQL/MX Applications and Modules

SQL/MX Release 3.1 provides two methods of compiling embedded SQL COBOL programs and creating modules. Both methods create an identical module file. The first method described, using embedded module definitions, is the default and preferred method.

The SQL/MX preprocessor reads a source file that contains COBOL and embedded SQL statements and generates:

Method 1: Embedded module definition	One file: a single, self-contained annotated source file that contains source statements with SQL statements converted to comments and embedded module definitions. You compile this file (<i>source-file.ecob</i> in embedded SQL/MX COBOL programs) with the COBOL compiler (<i>ecobol</i> or <i>nmcobol</i>) and the SQL/MX compiler (<i>mxCompileUserModule</i>). This is the default and preferred method.
Method 2: Annotated source file and module definition file	Two files: an annotated source file and a module definition file (<i>source-file.m</i>) that contains SQL source statements. You compile the source file with the COBOL compiler, and you compile the module definition file with the SQL/MX compiler (<i>mxcmp</i>). A module definition file is not created unless you use the <i>-x</i> or <i>-m</i> preprocessor options or set the <code>SQLMX_PREPROCESSOR_VERSION=800</code> environment variable to create a module definition file. For more information, see Influencing Module Management Behavior on page 17-9.

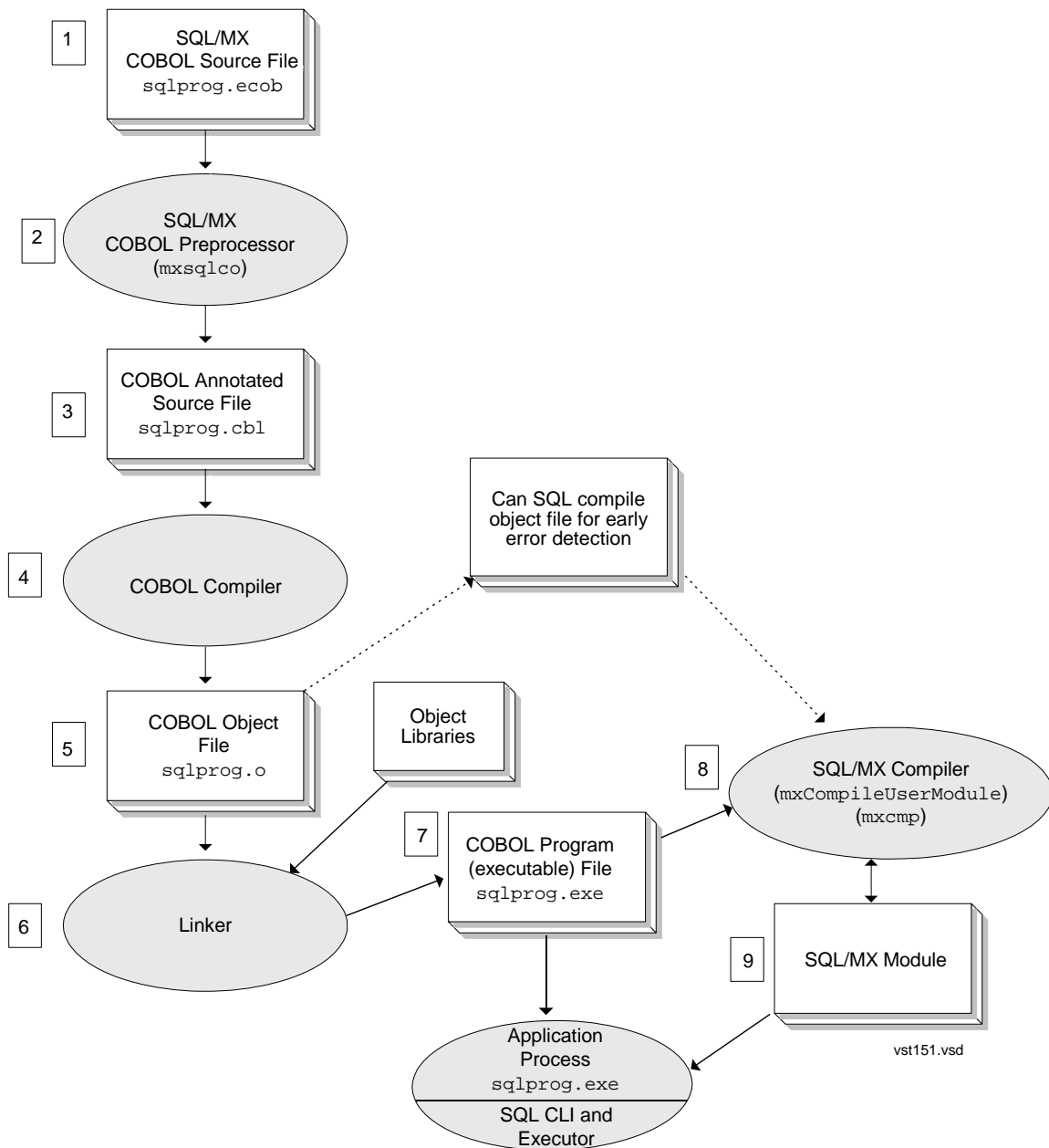
The preprocessor (*mxsqlco*) appends an embedded module definition, which includes a `?COLUMNS` directive by default, at the end of an annotated source file. Only one `?COLUMNS` directive is allowed in a source file. Otherwise, the COBOL compiler returns an error. If your source file contains a `?COLUMNS` directive, you must either:

- Remove the `?COLUMNS` directive from the source file before generating an embedded module definition.
- Keep the `?COLUMNS` directive in the source file and generate a module definition file instead of an embedded module definition.

Compiling Embedded SQL COBOL Programs With Embedded Module Definitions

Figure 16-1 shows how a self-contained, single-file COBOL program is compiled using embedded module definitions. The application's embedded SQL source file is called `sqlprog.ecob`.

Figure 16-1. Compiling Embedded SQL COBOL Programs With Embedded Module Definitions



Although this figure shows individual steps for clarity, you can use the COBOL compiler utility (`ecobol` or `nmcobol`) or the HP Enterprise Toolkit—NonStop Edition (ETK) to

automate the process. For information on using `ecobol` or `nmcobol` in this way, see [ecobol or nmcobol Utility: Using One Command for All Compilation Steps](#) on page 16-33. For more information on using ETK, see ETK online help.

These steps correspond to the steps in [Figure 16-1](#) on page 16-3.

1. Create the COBOL source file that contains embedded SQL statements (`sqlprog.ecob`).
2. Preprocess the application's embedded SQL source file by using the SQL/MX COBOL preprocessor `mxsqlco`. See [Running the SQL/MX COBOL Preprocessor](#) on page 16-9.

```
mxsqlco sqlprog.ecob
```

In this step, set optional module specification strings and `moduleCatalog` and `moduleSchema` default settings by using the `-g` option. See page [16-16](#) or [16-21](#). Although you do not set `mxcmp` defaults here, if the input source file contains `mxcmp` default settings, such as `EXEC SQL DECLARE/SET/CONTROL QUERY DEFAULT` statements, they are preprocessed into corresponding module language statements in the output module definition of the annotated source file.

3. The preprocessor produces a modified (annotated) COBOL source file (`sqlprog.cbl`) that contains the COBOL and SQL call-level interface (CLI) translations of embedded SQL statements and additional COBOL source constructs that represent the module definition. The default behavior creates a single, self-contained application source file with embedded module definitions.
4. Compile the annotated COBOL source file by using the `ecobol` or `nmcobol` compiler (OSS environment) or ETK (Windows environment). To produce an object file:

```
ecobol -Wcobol="consult /usr/tandem/sqlmx/lib/esqlcli.o" \
-o sqlprog.o -c sqlprog.cbl
```

```
nmcobol -Wcobol="consult /usr/tandem/sqlmx/lib/sqlcli.o" \
-o sqlprog.o -c sqlprog.cbl
```

If you do not specify the `-Wsqlmx` or `-Wmxcmp` flag in the command line, the `ecobol` or `nmcobol` compiler requires the `CONSULT` directive to compile the annotated COBOL source file correctly. The `esqlcli.o` or `sqlcli.o` file contains definitions of the CLI procedure calls for the translated SQL statements in the annotated COBOL source file. If you invoke `ecobol` or `nmcobol` with the `-Wsqlmx` or `-Wmxcmp` flag, the list of libraries searched automatically includes `esqlcli.o` or `sqlcli.o`.

Specify the `-c` option if you do not want `ecobol` or `nmcobol` to link the program. Otherwise, `ecobol` or `nmcobol` invokes `eld` or `nld` to create an executable file.

See [Running the COBOL Compiler and Linker](#) on page 16-23.

5. The COBOL compiler produces the object file, `sqlprog.o`. If you prefer early detection of SQL compilation errors, you can SQL compile the application's object file at this point. During program development, you might want to use the

`mxCompileUserModule` utility against all the object files rather than against the executable file. When you SQL compile against the object files, NonStop SQL/MX does not recompile each module for object files that are linked into more than one executable file.

6. Link application object files with object libraries to create an executable file by either:

- Running `ecobol` or `nmcobol` with object files as input to link them:

```
ecobol -o sqlprog.exe -lzclidll sqlprog.o
```

```
nmcobol -o sqlprog.exe -lzclisrl sqlprog.o
```

- Running the `eld` or `nld` utility separately after compilation to resolve external references in ENTER statements and implicit invocations of COBOL run-time library routines that many COBOL statements cause:

```
eld -lzcobdll -lzcresdll -lzclidll -o sqlprog.exe sqlprog.o
```

```
nld -lzcobsrl -lzcresrl -lzclisrl -o sqlprog.exe sqlprog.o
```

ZCLIDLL or ZCLISRL is a system library of the SQL/MX executor. You must specify this library in the command line if you invoke the linker, either by running `eld` or `nld` or by running `ecobol` or `nmcobol` without the `-Wsqlmx` or `-Wmxcmp` flag. If linking occurs when you invoke `ecobol` or `nmcobol` with the `-Wsqlmx` or `-Wmxcmp` flag, the list of libraries searched automatically includes ZCLIDLL or ZCLISRL.

Note. If you compiled a TNS/R native program with `-Wcall_shared` or `-Wshared`, you must link it with the `ld` utility instead of the `nld` utility.

7. The linker produces the application's executable file, `sqlprog.exe`.
8. SQL compile one, some, or all of the application's embedded module definitions in the executable file by using `mxCompileUserModule`. See [Running the SQL/MX Compiler](#) on page 16-25 and [Compiling Embedded Module Definitions](#) on page 16-25.

```
mxCompileUserModule sqlprog.exe
```

9. The SQL/MX compiler produces the SQL/MX module. The module is stored in the local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.

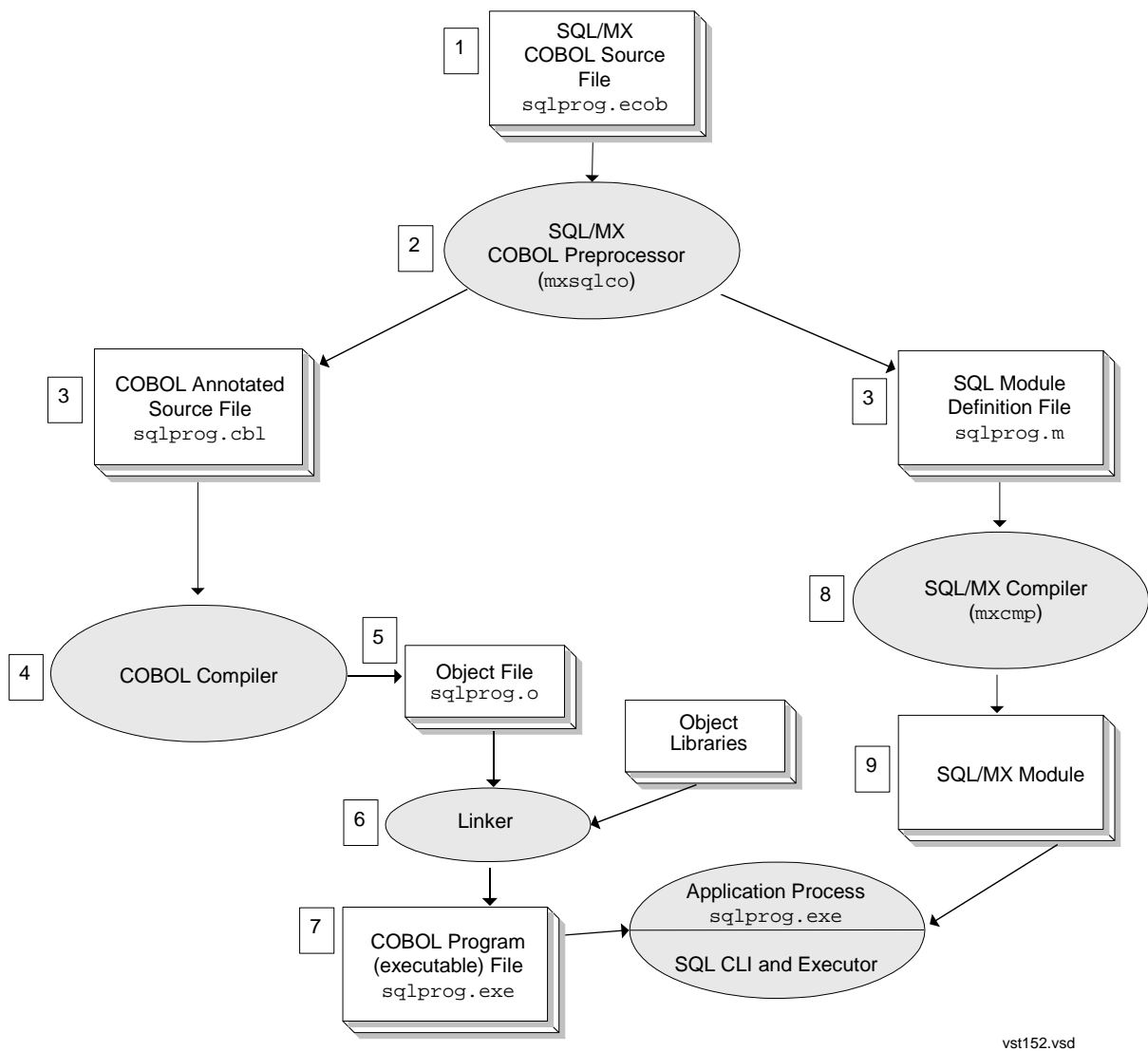
Run the COBOL executable program.

Note. The preprocessed COBOL source files are required for debugging. Neither `ecobol` nor `nmcobol` has an equivalent to the C/C++ `#line` directive.

Compiling Embedded SQL COBOL Programs With Module Definition Files

[Figure 16-2](#) shows how a COBOL program with separate module definition files is compiled. The application's embedded SQL source file is called `sqlprog.ecob`.

Figure 16-2. Compiling Embedded SQL COBOL Programs With Module Definition Files



Although this figure shows the individual steps for clarity, you can use the COBOL compiler utility (`ecobol` or `nmcobol`) or ETK to automate the process. For more information on using `ecobol` or `nmcobol` in this way, see [ecobol or nmcobol Utility: Using One Command for All Compilation Steps](#) on page 16-33. For more information on ETK, see ETK online help.

These steps correspond to the steps in [Figure 16-2](#) on page 16-6.

1. Create the COBOL source files that contain embedded SQL statements (`sqlprog.ecob`).
2. Preprocess the application's embedded SQL source files by using the SQL/MX COBOL preprocessor `mxsqlco`. See [Running the SQL/MX COBOL Preprocessor](#) on page 16-9.

```
mxsqlco sqlprog.ecob -c sqlprog.cbl -m sqlprog.m
```

In this step, set optional module specification strings and `moduleCatalog` and `moduleSchema` default settings by using the `-g` option. See page [16-16](#) or [16-21](#). Although you do not set `mxcmp` defaults here, if the input source file contains `mxcmp` default settings, such as `EXEC SQL DECLARE/SET/CONTROL QUERY DEFAULT` statements, they are preprocessed into corresponding module language statements in the output module definition of the module definition file. The preprocessor options (`-x` or `-m`) and the `SQLMX_PREPROCESSOR_VERSION=800` environment variable indicate to the preprocessor that you are compiling your program with module definition files. For more information on setting the preprocessor options, see [Module Management Behavior](#) on page 17-8.

3. The preprocessor produces two files: (1) a modified (annotated) COBOL source file (`sqlprog.cbl`) that contains the COBOL and SQL CLI translations of embedded SQL statements and (2) a module definition file (`sqlprog.m`).
4. Compile the annotated COBOL source file by using the `ecobol` or `nmcobol` compiler (OSS environment) or ETK (Windows environment). To produce an object file:

```
ecobol -Wcobol="consult /usr/tandem/sqlmx/lib/esqlcli.o" \
-o sqlprog.o -c sqlprog.cbl
```

```
nmcobol -Wcobol="consult /usr/tandem/sqlmx/lib/sqlcli.o" \
-o sqlprog.o -c sqlprog.cbl
```

If you do not specify the `-Wsqlmx` or `-Wmxcmp` flag in the command line, the `ecobol` or `nmcobol` compiler requires the `CONSULT` directive to compile the annotated COBOL source file correctly. The `esqlcli.o` or `sqlcli.o` file contains definitions of the CLI procedure calls for the translated SQL statements in the annotated COBOL source file. If you invoke `ecobol` or `nmcobol` with the `-Wsqlmx` or `-Wmxcmp` flag, the list of libraries searched automatically includes `esqlcli.o` or `sqlcli.o`.

Specify the `-c` option if you do not want `ecobol` or `nmcobol` to link the program. Otherwise, `ecobol` or `nmcobol` invokes `eld` or `nld` to create an executable file.

See [Running the COBOL Compiler and Linker](#) on page 16-23.

5. The COBOL compiler produces the object file, `sqlprog.o`.
6. Link application object files with object libraries to create an executable file by either:

- Running `ecobol` or `nmcobol` with object files as input to link them:

```
ecobol -o sqlprog.exe -lzclidll sqlprog.o
```

```
nmcobol -o sqlprog.exe -lzclisrl sqlprog.o
```

- Running the `eld` or `nld` utility separately after compilation to resolve external references in ENTER statements and implicit invocations of COBOL run-time library routines that many COBOL statements cause:

```
eld -lzcobdll -lzcresdll -lzclidll -o sqlprog.exe sqlprog.o
```

```
nld -lzcobsrl -lzcresrl -lzclisrl -o sqlprog.exe sqlprog.o
```

ZCLIDLL or ZCLISRL is a system library of the SQL/MX executor. You must specify this library in the command line if you invoke the linker, either by running `eld` or `nld` or by running `ecobol` or `nmcobol` without the `-Wsqlmx` or `-Wmxcmp` flag. If linking occurs when you invoke `ecobol` or `nmcobol` with the `-Wsqlmx` or `-Wmxcmp` flag, the list of libraries searched automatically includes ZCLIDLL or ZCLISRL.

Note. If you compiled a TNS/R native program with `-Wcall_shared` or `-Wshared`, you must link it with the `ld` utility instead of the `nld` utility.

7. The linker produces the application's executable file, `sqlprog.exe`.
8. SQL compile the application's module definition file by using the SQL/MX compiler (`mxcmp`). See [Running the SQL/MX Compiler](#) on page 16-25 and [Compiling a Module Definition File](#) on page 16-30.

```
mxcmp sqlprog.m
```

9. The SQL/MX compiler compiles the SQL source statements from the module definition file in a module 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. The module is stored in the local application directory, user-specified Guardian or OSS location or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.

Run the COBOL executable program.

Note. The preprocessed COBOL source files are required for debugging. Neither `ecobol` nor `nmcobol` has an equivalent to the C/C++ `#line` directive.

Creating Modules: From Development to Production

While HP recommends that you use embedded module definitions to create SQL modules, you might find it easier for debugging purposes to use module definition files during early stages of development and then switch to embedded module definitions upon deployment of your production system. Consider this:

- With embedded module definitions, you must successfully compile the output from the preprocessor before you can SQL compile the embedded module definition. You must diagnose host language errors in the source program before you can diagnose SQL errors in the source program.
- With module definition files, you compile the source file and the module definition file at the same time. This method provides the opportunity to diagnose both host language errors and SQL errors in the source file concurrently.

Embedded module definitions provide greater efficiency in the deployment of an application to a production environment.

Running the SQL/MX COBOL Preprocessor

The SQL/MX COBOL preprocessor is available for these environments:

- OSS
- Microsoft Windows
- Enterprise Plugins for Eclipse (EPE)
- Enterprise ToolKit—NonStop Edition (ETK)

Note. ETK is a GUI-based extension package to the Visual Studio .NET product. Use ETK to edit, compile, build, and deploy applications written in a variety of programming languages and with embedded SQL/MX. For more information, see ETK online help.

The preprocessor for the OSS environment is installed when you install NonStop SQL/MX on your system. You must install the Windows-hosted preprocessor on your Windows workstation. For information, see the *SQL/MX Release 3.2 Installation and Upgrade Guide*.

The syntax for using the preprocessor in each environment appears in [Syntax for the OSS-Hosted SQL/MX COBOL Preprocessor](#) on page 16-14 and [Syntax for the Windows-Hosted SQL/MX COBOL Preprocessor](#) on page 16-19.

Preprocessor Functions

The preprocessor processes COBOL and SQL source statements.

COBOL Source Statements

The preprocessor writes each COBOL source statement to the COBOL annotated source file. The preprocessor parses the COBOL annotated source file only to the extent necessary to detect scoping levels, host variable declarations, host variable expressions, and embedded SQL statements. The OSS-hosted SQL/MX COBOL preprocessor resolves the Guardian DEFINE used with a ?SOURCE *filename* directive and processes the file only if the -O command-line option is specified.

For example, consider the following source code:

```
?SOURCE =cobdef1
```



```
?SOURCE =cobdef2 (section1)
```

The OSS-hosted SQL/MX COBOL preprocessor resolves the `DEFINE =cobdef1` and the file mapped by `DEFINE` is processed if the `-O` command-line option is specified. Similarly, the file mapped by the `DEFINE =cobdef2` is processed for section `section1`.

The `DEFINE` used with a COBOL directive must be `MAP DEFINE`. The Windows-hosted COBOL preprocessor does not support Guardian `DEFINES`.

COBOL Comments

The preprocessor ignores COBOL comments unless the comment specifies a name for an SQL statement. You can use a comment to name an SQL statement explicitly. To do so, precede the statement with a comment using this format:

```
* SQL_statement_name = name [ comment-text ]
   EXEC SQL sql_statement ... END-EXEC.
```

The *name* is an SQL identifier you are assigning as the name of *sql_statement*, and *comment-text* is an optional comment that does not affect the assignment of the name. The COBOL comment must use only one line and must immediately precede the SQL statement.

For example, this comment names the SQL statement (`INSERT`) and provides comment text (“insert ten rows”):

```
* SQL_statement_name= INSERT insert ten rows
   EXEC SQL INSERT INTO ... END-EXEC.
```

If you do not specify a name for an SQL statement, the preprocessor assigns the statement a name of the form `SQLMX_DEFAULT_STATEMENT_n`, where *n* is an integer incremented by the preprocessor.

Host Variable Declarations

The preprocessor checks each host variable declaration (that is, a variable declared between `BEGIN DECLARE SECTION` and `END DECLARE SECTION`) to ensure that the variable uses a valid data type. For valid host-variable data types, see [Table 4-1](#) on page 4-5 and [Table 4-2](#) on page 4-7.

The preprocessor returns an error for embedded SQL statements that are not valid within a host-variable declaration section.

`SQLSTATE` must be declared within a Declare Section. See [Declaring SQLSTATE](#) on page 13-2.

Executable SQL Statements

The preprocessor performs these functions:

- Scans the statement for host variables (indicated by a colon) and ensures that each host variable is declared within the current scope of the program.
- Converts the SQL statement to a COBOL comment in the COBOL annotated source file.
- Writes data structure initialization statements needed for arguments to the CLI procedure calls and writes the appropriate CLI procedure call or calls for the SQL statement immediately after the commented statement in the COBOL annotated source file. At run time, the calls invoke the SQL/MX executor to execute the procedure for the SQL statement within the module.
- Writes the executable SQL statement to a separate module definition file if you use the `-x` or `-m` preprocessor option or set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable.

Use the preprocessor to embed SQL anywhere in the COBOL source file. However, the preprocessor determines in which part of the source file the embedded SQL is located and issues warnings if an embedded SQL statement is not placed correctly. See [Placement of SQL Statements](#) on page 2-2.

At the end of processing the embedded SQL COBOL source file, the preprocessor checks the status of static cursors:

- Cursors accessed and not opened return an error message.
- Cursors declared and not accessed return a warning message.

Preprocessor Output

COBOL Annotated Source File for Embedded Module Definitions

The SQL/MX COBOL preprocessor processes a COBOL source file, such as *source-file.ecob*, and generates one COBOL annotated source file (*source-file.cbl*) as its output file. The annotated source file contains the embedded module definitions.

COBOL Annotated Source File for Module Definition Files

If you use the `-x` or `-m` preprocessor option or if you set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable, the preprocessor processes a COBOL source file, such as *source-file.ecob*, and generates two files: the annotated source file (*source-file.cbl*) and the module definition file (*source-file.m*).

For more information on module management behavior and influencing the preprocessor, see [Module Management Behavior](#) on page 17-8. For recommended naming conventions for COBOL source files, see [Table 17-1](#) on page 17-1.

The preprocessor converts the embedded SQL statements to COBOL comments, followed by the appropriate CLI calls.

The COBOL annotated source file consists of:

Header	Contains the data structures.
Body	Contains the embedded SQL COBOL source file translated into COBOL statements. The preprocessor converts each embedded SQL statement to a COBOL comment by prefixing an asterisk (*) to the statement and follows the commented statement with a CLI call that invokes the executor at run time to execute the statement.
Trailer	Contains definitions required to complete the COBOL source file. Definitions include the module version number, the creation timestamp (the operating system timestamp when the preprocessor was invoked), and the module name.

Header for Module Definition File

If you specify the `-m` or `-x` preprocessor option or set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable, the preprocessor creates a module definition file in your current directory that contains embedded SQL statements. The preprocessor writes the header of the module definition file as:

```
MODULE module-name NAMES ARE ISO88591 ;
TIMESTAMP DEFINITION ( creation_timestamp ) ;
source-file 'source-file location' ;
```

You can specify *module-name* by using the MODULE directive as the beginning statement in the PROCEDURE DIVISION of your embedded SQL COBOL program. For example:

```
EXEC SQL MODULE EXF62M NAMES ARE ISO88591 END-EXEC.
```

Otherwise, if you do not specify a MODULE directive, the preprocessor generates a system-supplied module name for you. See also the MODULE directive in the *SQL/MX Reference Manual*.

Trailer for Annotated-Source File

The *module-name* and the *creation_timestamp* correspond to these same elements in the trailer of the COBOL source file. The SQL/MX compiler uses *module-name* to name the module file. It also writes the *creation_timestamp* into the module file. The COBOL source file is then compiled and linked. When the resulting program file is executed and calls the SQL/MX executor, the preprocessor-generated CLI procedure calls pass the *module-name* and *creation_timestamp* to the executor. The executor uses the *module-name* to locate the corresponding module file. The *creation_timestamp* is used to ensure that the version of the executable program is synchronized with the version of the module file. This strategy prevents, for example, the executable program from being altered and rebuilt without rebuilding the module file. For more information, see [Understanding and Avoiding Common Run-Time Errors](#) on page 16-52.

The ISO88591 character set is the default character set for CHAR or VARCHAR data types for NonStop SQL/MX.

Procedures

After writing to the header of the module definition file, the preprocessor writes procedures for executing SQL statements. A procedure consists of a name, a formal argument list, and an SQL statement as the body of the procedure.

Each formal argument has a name and an SQL data type. The arguments are the host variables that occur in the SQL statement in the body of the procedure. The preprocessor writes the arguments in the same order as the first occurrence of the host variables, scanning from left to right, in the SQL statement. In some cases, the arguments are data structures that contain references to host variables. The host variable references are stored in the same order in which they appear in the SQL statement.

OSS-Hosted SQL/MX COBOL Preprocessor

You can compile and run an embedded SQL COBOL program in the OSS environment on a NonStop system. Although you cannot compile and run such a program in the Guardian environment, you can use an OSS pass-through command in the Guardian environment. For instructions on using the Windows-hosted SQL/MX COBOL preprocessor, see [Windows-Hosted SQL/MX COBOL Preprocessor](#) on page 16-18. For instructions on using the OSS pass-through command to execute the preprocessor in the Guardian environment, see [Building SQL/MX Guardian Applications in the Guardian Environment](#) on page 16-47.

The OSS-hosted SQL/MX COBOL preprocessor (`mxsqlco`) is installed in the `/usr/tandem/sqlmx/bin` directory in the OSS environment. You can use the `ecobol` or `nmcobol` utility to preprocess embedded SQL COBOL programs, compile COBOL and run the SQL/MX compiler, and then link the COBOL program. For more information, see [ecobol or nmcobol Utility: Using One Command for All Compilation Steps](#) on page 16-33.

Syntax for the OSS-Hosted SQL/MX COBOL Preprocessor

```

mxsqlco sql-file
[ -c COBOL-output-file ]
[ -m module-def-file ]
[ -e ]
[ -l list-file ]
[ -a ]
[ -f ]
[ -t timestamp ]
[ -q ]
[ -d toggle || SETTOG ]
[ -x ]
[ -g {moduleGroup[=module-group-specification-string]
      |moduleTableSet[=module-tableset-specification-
      string]
      |moduleVersion[=module-version-specification-
      string]
      |moduleCatalog[=module-catalog-name]
      |moduleSchema[=module-schema-name]
      } ]
[ -Q { [invokeCatalog=catalog-name]
      | [invokeSchema=schema-name]
      } ]
[ -O ]

```

sql-file

is the name of the input COBOL source file that contains embedded SQL statements.

-c COBOL-output-file

is the name of the output preprocessed annotated source file that contains COBOL statements and embedded SQL statements converted to comments. This file is the input for the COBOL compiler (*ecobol* or *nmcobol* utility). The default is *source-file.cbl*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

-m module-def-file

is the name of the output module definition file, which is the input file for the SQL/MX compiler. The default is *source-file.m*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

-e

generates CHARACTER data types for date-time data types. This behavior is compatible with NonStop SQL/MX Release 1.8. For more information, see [INVOKE](#)

[and Date-Time and Interval Host Variables \(SQL/MX Release 1.8 Applications\)](#) on page 4-23.

-l *list-file*

is the name of the output list file that contains preprocessor error and warning messages. The default is *source-file.lst*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

-a

indicates the ANSI fixed format for the source file. Output source is in the same format. If not specified, -f is the default.

-f

indicates the TANDEM free format for the source program. Output source is in the same format. If not specified, -f is the default.

-t *timestamp*

provides a creation timestamp that the preprocessor writes to the COBOL annotated source file (and the module definition file if the -x or -m preprocessor option or the `SQLMX_PREPROCESSOR_VERSION=800` environment variable is used). The *timestamp* value overrides the operating system timestamp.

For example, you can specify these timestamp values:

```
-t "2005-10-26 09:01:20.00"
-t 2005-10-26.12:0000.000000
```

The preprocessing timestamp of the generated code must match the preprocessing timestamp stored in the module. Use this option with caution and only when you need to change the source text of the embedded SQL program without SQL-compiling the generated code.

-q

directs the preprocessor to accept SQL string literals delimited by double quotes in addition to single quotes. If you specify -q, you cannot use SQL delimited identifiers.

-d *toggle* || SETTOG

defines toggles for use with conditional compilation. Toggles must be in the range of 1 through 15. If you specify SETTOG, all toggles are set to ON.

-x

directs the preprocessor to refrain from emitting embedded module definitions into the annotated output source file.

```
-g { moduleGroup[=module-group-specification-string]
    moduleTableSet[=module-tableset-specification-string]
    moduleVersion[=module-version-specification-string]
    moduleCatalog[=module-catalog-name]
    moduleSchema[=module-schema-name]
}
```

specifies the arguments for qualifying the name given to the compiled module file. If you use this option, you must supply at least one of the five module management attributes. If you want to specify more than one attribute, repeat the entire `-g` option for each attribute. These attribute values are used to qualify the name of the compiled module file. For more information, see [File Naming Conventions](#) on page 17-1.

To use the `-g` option, you must supply a value in conjunction with the `moduleGroup`, `moduleTableSet`, `moduleVersion`, `moduleCatalog`, or `moduleSchema` attribute. The value must immediately follow the equal sign, and the equal sign must immediately follow the attribute keyword. The value can use regular or delimited identifiers. (See the description of regular and delimited identifiers in the *SQL/MX Reference Manual*.) If you supply more than one value for any attribute, only the final value is used. For information on the length of the module name, see [Module Name Length](#) on page 17-12.

`moduleGroup`

sets the `moduleGroup` attribute to group an application's module files logically by sharing the same name prefix. The `moduleGroup` becomes embedded in the module file names as a common prefix and enables the use of OSS wild-card file specification patterns to manage the files. For more information, see [Grouping](#) on page 17-23. The maximum size for the `moduleGroup` attribute is 31 characters.

`moduleTableset`

sets the `moduleTableSet` attribute to use the module management targeting feature. You can create different sets of module files that can be used against different sets of tables. For more information, see [Specifying the search locations of the module files](#) on page 17-13. The maximum size for the `moduleTableSet` attribute is 31 characters.

`moduleVersion`

sets the `moduleVersion` attribute to enable multiple versions of an application's module files to coexist while keeping the same `MODULE` directive in each version. For more information, see [Versioning](#) on page 17-21. The maximum size for the `moduleVersion` attribute is 31 characters.

`moduleCatalog`

sets the `moduleCatalog` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a catalog name. If

the `moduleCatalog` option is not set, the preprocessor emits the output `MODULE` directive using the default catalog naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleCatalog` attribute is 128 characters.

`moduleSchema`

sets the `moduleSchema` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a schema name. The `moduleSchema` can contain a catalog name. If the `moduleSchema` attribute is not used, the preprocessor emits the output `MODULE` directive by using the default schema naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleSchema` attribute is 128 characters.

```
[ -Q { [ invokeCatalog=catalog-name ]
      [ invokeSchema=schema-name ]
    } ]
```

specifies the catalog name and schema name qualifiers for objects inside the `invoke` clause. If you use this option, specify one of the attributes—`invokeCatalog` or `invokeSchema`. If you want to specify both the attributes, repeat the `-Q` option for each attribute.

`invokeCatalog`

sets the catalog for unqualified objects inside the `invoke` clause as *catalog-name*. If a catalog is specified using the Control Query Default Catalog or Declare Catalog, this attribute has no effect. The maximum size of the `invokeCatalog` attribute is 128 characters.

`invokeSchema`

sets the schema for unqualified objects inside the `invoke` clause as *schema-name*. If a schema is specified using the Control Query Default Schema or Declare Schema, this attribute has no effect. The maximum size of the `invokeSchema` attribute is 128 characters.

```
[ -O ]
```

replaces Guardian `DEFINE` in the `?SOURCE` directive, in the OSS file format. The `DEFINEs` are resolved only if the preprocessor option `-O` is specified.

Examples—mxsqlco

Run the SQL/MX COBOL preprocessor using the `mxsqlco` command. This example creates an annotated source file and module definition file:

```
mxsqlco sqlprog.ecob -c sqlprog.cbl -m sqlprog.m -g \
moduleTableSet=T1
```

This example creates a self-contained, annotated output source file that contains an embedded module definition:

```
mxsqlco sqlprog.ecob -c sqlprog.cbl \
-g moduleGroup=INVENTORY -g moduleVersion=V2
```

Windows-Hosted SQL/MX COBOL Preprocessor

The Windows-hosted SQL/MX COBOL preprocessor is a DLL file named `mxsqlcont.dll` and is accompanied by a DLL loader named `mxsqlco.exe`. These files are installed in the `C:\Program Files\HP SQL-MX COBOL Preprocessors` directory. Use either the command shell or the Korn shell to run the preprocessor with the `RUN` command.

You can install multiple versions of the SQL/MX COBOL preprocessors. The environment variable `MXSQLCO` enables you to select a particular version of the SQL/MX COBOL preprocessor for COBOL compilations.

For example, to select the SQL/MX COBOL preprocessor in the `C:\PROGRA~1\HPSQL-~2\` directory, set `MXSQLCO` from the Windows command line:

```
set MXSQLCO=C:\PROGRA~1\HPSQL-~2\mxsqlcont.dll
```

You can also set the environment variable in the Windows system properties. If multiple versions of the SQL/MX COBOL preprocessors are installed and if `MXSQLCO` is not set, the latest version of the SQL/MX COBOL preprocessor installed on the system is used for compilations, by default.

Note. On systems running H06.14 RVU and later, the SQL/MX COBOL compilations select the latest version of the SQL/MX COBOL preprocessors installed on the system.

On systems running H06.13 RVU and earlier, the SQL/MX COBOL compilations use the earliest version of the SQL/MX COBOL preprocessor installed on the system.

If you use `INVOKE`, `MXCS` must be installed on your operating system to provide the necessary communication between your client workstation and the server. For more information on how to install `MXCS`, see the *SQL/MX Connectivity Service Administrative Command Reference*. In addition, you must install the HP NonStop ODBC/MX driver for Windows. For installation information, see the *ODBC/MX Driver for Windows Manual*.

The SQL/MX COBOL preprocessor 3.1 can be invoked by a user, a cross compiler, or by an IDE. For each scenario, the SQL preprocessor invoked is:

- A user calls the SQL preprocessor to preprocess a source program. The SQL preprocessor uses the header files and libraries from the SQL preprocessor installation directory.
- The SQL preprocessor is invoked by the `nmcobol.exe/ecobol.exe` compiler. The compiler uses the SQL preprocessor version defined by `MXSQLCO`. The SQL preprocessor uses the libraries and header files related to that version.

If `MXSQLCO` is not set, the cross compiler invokes the latest version of the SQL/MX COBOL preprocessor installed on the system.

- An IDE is used. The IDE invokes `c89`, which uses `MXSQLCO` to select an alternative version. Some of the IDEs are:
 - Enterprise Tool Kit (ETK)—plug-in for Microsoft Visual studio

The environment variable `MXSQLCO` must be set before starting ETK.
 - Enterprise Plugins for Eclipse (EPE)—plug-in for Eclipse

When Eclipse is used, `MXSQLCO` is set by EPE based on the value of the preprocessor installation location.

Syntax for the Windows-Hosted SQL/MX COBOL Preprocessor

```

mxsqlco sql-file
      [ -c COBOL-output-file ]
      [ -m module-def-file ]
      [ -e ]
      [ -l list-file ]
      [ -a ]
      [ -f ]
      [ -t timestamp ]
      [ -q ]
      [ -d toggle || SETTOG ]
      [ -s system-name or IP-address ]
      [ -r ODBC-listener ]
      [ -y NSK-username ]
      [ -z NSK-password ]
      [ -x ]
      [ -g {moduleGroup[=module-group-specification-string]
            |moduleTableSet[=module-tableset-specification-string]
            |moduleVersion[=module-version-specification-string]
            |moduleCatalog[=module-catalog-name]
            |moduleSchema[=module-schema-name]
          } ]
      [ -Q { [invokeCatalog=catalog-name]
            | [invokeSchema=schema-name]
          } ]

```

sql-file

is the name of the input COBOL source file that contains embedded SQL statements.

`-c COBOL-output-file`

is the name of the output preprocessed annotated source file that contains COBOL statements and embedded SQL statements converted to comments. This file is the input file for the COBOL compiler. The default is *source-file.cbl*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

`-m module-def-file`

is the name of the output module definition file, which is the input file for the SQL/MX compiler. The default is *source-file.m*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

`-e`

generates CHARACTER data types for date-time data types. This behavior is compatible with NonStop SQL/MX Release 1.8. For more information, see [INVOKE and Date-Time and Interval Host Variables \(SQL/MX Release 1.8 Applications\)](#) on page 4-23.

`-l list-file`

is the name of the output list file that contains preprocessor error and warning messages. The default is *source-file.lst*, where *source-file* is the name of the SQL/MX COBOL source file (for example, *sqlprog.ecob*) without the file extension.

`-a`

indicates the ANSI fixed format for the source file. Output source is in the same format. If not specified, `-f` is the default.

`-f`

indicates the TANDEM free format for the source program. Output source is in the same format. If not specified, `-f` is the default.

`-t timestamp`

provides a creation timestamp that the preprocessor writes to the COBOL annotated source file (and the module definition file if the `-x` or `-m` preprocessor option or the `SQLMX_PREPROCESSOR_VERSION=800` environment variable is used). The *timestamp* value overrides the operating system timestamp.

The preprocessing timestamp of the generated code must match the preprocessing timestamp stored in the module. Use this option with caution and only when you need to change the source text of the embedded SQL program without SQL-compiling the generated code.

-q

directs the preprocessor to accept SQL string literals delimited by double quotes in addition to single quotes. If you specify -q, you cannot use SQL delimited identifiers.

-d *toggle* || SETTOG

defines toggles for use with conditional compilation. Toggles must be in the range of 1 through 15. If you specify SETTOG, all toggles are set to ON.

-s *system-name* or *IP-address*

is the node name or IP address of the NonStop system where the tables are found by INVOKE. This option is required if you use INVOKE.

-r *ODBC-listener*

is the NonStop system port to connect to for the ODBC listener process. The default port for the Association server is 18650.

-y *NSK-username*

is the Guardian user name with access to the tables that INVOKE reads. This option is required if you use INVOKE.

-z *NSK-password*

is the password for the user name for the NonStop system. This option is required if you use INVOKE.

-x

directs the preprocessor to refrain from emitting embedded module definitions into the annotated output source file.

```
-g {moduleGroup[=module-group-specification-string]
   {moduleTableSet[=module-tableset-specification-string]
   {moduleVersion[=module-version-specification-string]
   {moduleCatalog[=module-catalog-name]
   {moduleSchema[=module-schema-name]
   }
   }
   }
```

specifies the arguments for qualifying the name given to the compiled module file. If you use this option, you must supply at least one of the five module management attributes. If you want to specify more than one attribute, repeat the entire -g option for each attribute. These attribute values are used to qualify the name of the compiled module file. See [File Naming Conventions](#) on page 17-1.

To use the -g option, you must supply a value in conjunction with the moduleGroup, moduleTableSet, moduleVersion, moduleCatalog, or moduleSchema attribute. The value must immediately follow the equal sign, and the equal sign must immediately follow the attribute keyword. The value can use

regular or delimited identifiers. (See the description of regular and delimited identifiers in the *SQL/MX Reference Manual*.) If you supply more than one value for any attribute, only the final value is used. For information on the length of the module name, see [Module Name Length](#) on page 17-12.

`moduleGroup`

sets the `moduleGroup` attribute to group an application's module files logically by sharing the same name prefix. The `moduleGroup` becomes embedded in the module file names as a common prefix and enables the use of OSS wildcard file specification patterns to manage the files. For more information, see [Grouping](#) on page 17-23. The maximum size for the `moduleGroup` attribute is 31 characters.

`moduleTableset`

sets the `moduleTableSet` attribute to use the module management targeting feature. You can create different sets of module files that can be used against different sets of tables. For more information, see [Specifying the search locations of the module files](#) on page 17-13. The maximum size for the `moduleTableSet` attribute is 31 characters.

`moduleVersion`

sets the `moduleVersion` attribute to enable multiple versions of an application's module files to coexist while keeping the same `MODULE` directive in each version. For more information, see [Versioning](#) on page 17-21. The maximum size for the `moduleVersion` attribute is 31 characters.

`moduleCatalog`

sets the `moduleCatalog` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a catalog name. If the `moduleCatalog` option is not set, the preprocessor emits the output `MODULE` directive using the default catalog naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleCatalog` attribute is 128 characters.

`moduleSchema`

set the `moduleSchema` attribute if the input *sql-file* does not have a `MODULE` directive or its `MODULE` directive does not specify a schema name. The `moduleSchema` can contain a catalog name. If the `moduleSchema` attribute is not used, the preprocessor emits the output `MODULE` directive by using the default schema naming rules described in the *SQL/MX Reference Manual*. The maximum size for the `moduleSchema` attribute is 128 characters.

```
[ -Q { [ invokeCatalog=catalog-name ]
      | [ invokeSchema=schema-name ]
    } ]
```

specifies the catalog name and schema name qualifiers for objects inside the `invoke` clause. If you use this option, specify one of the attributes—`invokeCatalog` or `invokeSchema`. If you want to specify both the attributes, repeat the `-Q` option for each attribute.

`invokeCatalog`

sets the catalog for unqualified objects inside the `invoke` clause as *catalog-name*. If a catalog is specified using the Control Query Default Catalog or Declare Catalog, this attribute has no effect. The maximum size of the `invokeCatalog` attribute is 128 characters.

`invokeSchema`

sets the schema for unqualified objects inside the `invoke` clause as *schema-name*. If a schema is specified using the Control Query Default Schema or Declare Schema, this attribute has no effect. The maximum size of the `invokeSchema` attribute is 128 characters.

Examples—mxsqlco

Run the SQL/MX COBOL preprocessor using the `mxsqlco` command. This example creates an annotated source file and module definition file:

```
mxsqlco sqlprog.ecob -c sqlprog.cbl -m sqlprog.m -g \
moduleTableSet=T1
```

This example creates a single-file, annotated output source file that contains an embedded module definition:

```
mxsqlco sqlprog.ecob -c sqlprog.cbl \
-g moduleGroup=INVENTORY -g moduleVersion=V2
```

Running the COBOL Compiler and Linker

The HP NonStop COBOL compilers translate source code into machine language that is specific to a particular NonStop system architecture. The type of COBOL compiler that you use to compile your SQL/MX program determines the NonStop system and environment where you can run the program.

Note. TNS/R native compilation tools are available on systems running H06.05 or later RVUs.

[Table 16-1](#) lists the COBOL compilers, the environments where you can run the compilers, and the environments where you can run the compiled programs.

Table 16-1. HP NonStop COBOL Compilers for Embedded SQL/MX Programs

Compiler	Compiler Operating Environment	Program Execution Environment
TNS/E native compilers:		
● Native COBOL cross compiler for TNS/E*	Windows environment on a PC connected to a NonStop system running an H-series RVU	OSS or Guardian environment on a NonStop system running an H-series RVU
● <code>ecobol</code>	OSS environment on a NonStop system running an H-series RVU	OSS or Guardian environment on a NonStop system running an H-series RVU
● ECOBOL	Guardian environment on a NonStop system running an H-series RVU	Guardian or OSS environment on a NonStop system running an H-series RVU
TNS/R native compilers:		
● Native COBOL cross compiler for TNS/R*	Windows environment on a PC connected to a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	OSS or Guardian environment on a NonStop system running a G-series RVU
● <code>nmcobol</code>	OSS environment on a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	OSS or Guardian environment on a NonStop system running a G-series RVU
● NMCOBOL	Guardian environment on a NonStop system running H06.05 or later RVU or a NonStop system running a G-series RVU	Guardian or OSS environment on a NonStop system running a G-series RVU

* The native COBOL cross compilers can be run from the ETK or from the PC command line.

On Windows, you can run the COBOL compiler and native object file linker from ETK, or you can use the command-line cross compiler (`ecobol` or `nmcobol`) and the linker (`eld` or `nld`). For details on syntax and using the COBOL cross compiler with ETK, see the help file *Using Command-Line Cross Compilers on Windows*, which is included with ETK.

To run the ECOBOL compiler, see the *COBOL Manual for TNS/E Programs*. To run the `eld` linker, see the *eld Manual*. To run the NMCOBOL compiler, see the *COBOL Manual for TNS and TNS/R Programs*. To run the `nld` linker, see the *nld Manual*. For more information on the `ecobol` or `nmcobol` utility, see [ecobol or nmcobol Utility](#).

[Using One Command for All Compilation Steps](#) on page 16-33, the OSS reference pages, or the *Open System Services Shell and Utilities Reference Manual*.

Running the SQL/MX Compiler

The SQL/MX compiler compiles and optimizes static and dynamic SQL statements for subsequent execution by the SQL/MX executor and performs these functions:

- Expands SQL object names by using the current default settings
- Expands view definitions
- Performs type checking for COBOL and SQL data types
- Checks SQL object references to verify their existence
- Determines an optimized execution plan and access path for each DML statement if the SQL objects in the statement are present at compile time
- Generates executable code for the execution plans (if the SQL objects in the statement are present at compile time) and creates a module in the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory
- Generates a list of the SQL statements in the program file, including messages
- Returns a completion code indicating the outcome of the compilation

The SQL/MX compiler is an OSS program installed in the Guardian `$SYSTEM.SYSTEM` subvolume (`/G/system/system/` in the OSS environment). You must run the compiler in the OSS environment. It does not run as a Guardian process.

You must explicitly invoke the SQL/MX compiler to compile static SQL statements. At run time, the SQL/MX executor also invokes the compiler to compile dynamic SQL statements and to recompile any static SQL statements that refer to database objects that have changed and that affect the SQL statement's execution plan.

If your program accesses a table that has changed since the last static compilation, you should statically recompile the program to improve performance. Otherwise, NonStop SQL/MX dynamically recompiles the program before each execution.

Compiling Embedded Module Definitions

To compile one or more of the modules of an embedded SQL/MX application executable, use the `mxCompileUserModule` utility on the object file created by the COBOL compiler or on the executable file created by the linker.

If you have a combination of module definition files and applications that contain embedded module definitions, use `mxCompileUserModule` to SQL compile the self-contained object files containing embedded module definitions, and use `mxcmp` to SQL compile the application's separate module definition files. For more information, see

[Compiling a Module Definition File](#) on page 16-30 and [Combining Embedded Module Definitions and Module Definition Files](#) on page 16-46.

Command Line Syntax

To invoke `mxCompileUserModule`, at an OSS prompt, enter:

```
mxCompileUserModule { { [-e] [-v] [-g {moduleGlobal |
moduleLocal}]
[-d compiler-attribute-name=compiler-attribute-value]... } |
-m } Application-file ["{"module-name [, module-name]...""}"]

Module-name is:
[[Catalog.]Schema.]Module [MODULEGROUP=group]
[MODULETABLESET=target] [MODULEVERSION=version]
```

`-e`

directs `mxCompileUserModule` to generate a warning rather than an error if a table or class MAP DEFINE in an SQL statement does not exist during explicit SQL/MX compilation. To find errors in your program during explicit SQL/MX compilation, omit the `-e` option.

If you are using late name resolution and want to use a table or DEFINE that does not exist during explicit SQL/MX compilation, include the `-e` option. Then at run time, the SQL/MX executor automatically recompiles the SQL statement from the statement's source in the module by using the run-time version of the table.

`-v`

directs `mxCompileUserModule` to display summary information in addition to error and warning messages for the compilation. For example, use this option to verify the default settings of the SQL/MX compiler.

`-m`

directs `mxCompileUserModule` to display the list of module files associated with the application file.

Note.

- If `-m` option is specified, other command line options are ignored.
 - The OSS tool `mxCompileUserModule` with the `-m` option does not display the module files associated with the DLLs loaded by the embedded SQL executable.
 - `mxCompileUserModule` with the `-m` option does not display the list of module names associated with an application file if the modules are generated from a source SQL file using the `-x` preprocessor option or if the environment variable, `SQLMX_PREPROCESSOR_VERSION` is set to 800.
-

`-g moduleGlobal`

specifies that the module is placed globally in the `/usr/tandem/sqlmx/USERMODULES` directory.

`-g moduleLocal[=<OSSdir>]`

directs `mxCompileUserModule` to place the module in the OSS directory. The OSS directory can be either a Guardian or OSS location in the OSS format. If the OSS directory is omitted, the module is created in the current directory. The following rules related to the OSS directory apply:

- The OSS directory must exist and be accessible.
- The directory must not be a remote directory in an Expand network.
- The OSS directory must not exceed 1024 characters.

If these conditions are not met, an error is generated, and no module is created.

If you do not specify `-g moduleLocal[=<OSSdir>]` but set `MXCMP_PLACES_LOCAL_MODULES ON`, you must be in the same directory as the application executable when you invoke `mxCompileUserModule`. Otherwise, `mxCompileUserModule` writes the module in the current directory, and you will need to move the module to the global `USERMODULES` directory or co-locate the module with its application. For more information, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

`-d compiler-attribute-name=compiler-attribute-value`

specifies default attribute settings for compilation and existing settings for the module name. The module name settings are:

```
modulecatalog=cat
moduleschema=sch
modulegroup=grp
moduletableset=tgt
moduleversion=ver
```

The `-d modulecatalog`, `moduleschema`, `modulegroup`, `moduletableset`, and `moduleversion` options are similar to the `mxsqlco -g modulecatalog`, `moduleschema`, `modulegroup`, `moduletableset`, and `moduleversion` options because you use them to externally qualify simple module names. These options are not CONTROL QUERY DEFAULT settings (however, all other `-d attr=value` pairs are). In addition, there is no default value for `-d modulecatalog` or `-d moduleschema`.

The module name settings must match the module management options you specified during preprocessing. See [Running the SQL/MX COBOL Preprocessor](#) on page 16-9.

The default attribute settings for compilation override settings in the `SYSTEM_DEFAULTS` table but do not override the object name qualification or the

settings of embedded CONTROL QUERY DEFAULT, DECLARE, or SET statements, which are set in the input source file. For more information, see the SYSTEM_DEFAULTS table in the *SQL/MX Reference Manual*.

The OSS shell is used to invoke `mxCompileUserModule`, which in turn uses the OSS shell to invoke `mxcmp`. Consequently, you must adjust the syntax for setting CONTROL QUERY DEFAULT attribute values for `MP_SYSTEM` and `MP_VOLUME`. The OSS shell performs command/parameter substitution and allows a `\` (backslash) to quote special characters such as `$`.

This example shows how to set `MP_SYSTEM` and `MP_VOLUME` as `mxCompileUserModule` command-line options:

```
to get MP_SYSTEM=\KINGPIN    --> use -d MP_SYSTEM=\\KINGPIN
to get MP_VOLUME=$TX012      --> use -d MP_VOLUME=\\$TX012
```

application-file

is the OSS path name of an object file that contains embedded module definitions. The OSS directory:

- Must exist and be accessible. Otherwise, an error is returned, and no module is created.
- Must not specify a Guardian subvolume (`/G/...`) or a remote directory in an Expand network (`/E/...`).
- Must not exceed 1024 characters.

module-name

is the fully qualified name of an embedded module definition. This option names the generated module that is written to the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s) or to the global `/usr/tandem/sqlmx/USERMODULES` directory. For more information, see [Module Management Naming](#) on page 17-9.

Each *module-name* consists of:

```
[ [catalog.]schema. ]module [MODULEGROUP=group]
[MODULETABLESET=target] [MODULEVERSION=version]
```

If *catalog* and *schema* are omitted, their default value settings can be supplied with `-d MODULECATALOG=catalog` or `-d MODULESCHEMA=schema`. If `MODULEGROUP`, `MODULETABLESET`, or `MODULEVERSION` is omitted, the default setting can be supplied with `-d MODULEGROUP=group`, `-d MODULETABLESET=target`, or `-d MODULEVERSION=version`.

If no module name is specified, `mxCompileUserModule` operates on all embedded module definitions of *application-file*. Otherwise, each *module-name* is the fully qualified three-part name of an embedded module definition in *application-file*, and `mxCompileUserModule` operates only on the named embedded module definitions.

In summary, modules can be named as:

- A fully qualified delimited module name, such as
`cat.sch.\"GRP^MODULE^TGT^VER\"`
- A qualified module name followed by module specification strings, such as
`cat.sch.module MODULEGROUP=grp MODULETABLESET=tgt
MODULEVERSION=ver`
- A simple, unqualified module name (for example, `mod`), with the catalog, schema, group, table set, or version specified as `-d` compiler attributes.

You can run `mxCompileUserModule` more than once.

`mxCompileUserModule` extracts the *application-file*'s selected module definitions. For each selected module definition `m`, `mxCompileUserModule` passes `m` to `mxcmp` for SQL compilation. Each compilation of a selected module definition either succeeds or fails just like any `mxcmp` invocation. An `mxcmp` compilation failure does not affect preceding or following `mxcmp` invocations. In particular, an `mxcmp` compilation failure does not prevent `mxCompileUserModule` from proceeding with the `mxcmp` compilation of the next selected module definition.

Examples—mxCompileUserModule

- This command compiles the embedded module definition:

```
mxCompileUserModule sqlprog.exe
```

- This command places the module file in the same OSS directory as the application executable:

```
mxCompileUserModule -g moduleLocal sqlprog.o
```

- These settings affect statement recompilation at execution time:

```
mxCompileUserModule -d AUTOMATIC_RECOMPILATION=ON \  
-d SIMILARITY_CHECK=ON sqlprog.exe
```

- The following command compiles the embedded module definition and places the module file in the user-specified OSS location, `/usr/mymodules`:

```
mxcompileusermodule -g moduleLocal=/usr/mymodules sqlprog.exe
```

- The following command compiles the embedded module definition and places the module file in the user-specified Guardian location, `/G/data01/mymod`:

```
mxcompileusermodule -g moduleLocal=/G/data01/mymod  
sqlprog.exe
```

- The following command displays the list of module files associated with the file, CAT.SCH.TEST.EXE:

```
mxCompileUserModule -m CAT.SCH.TEST.EXE
```

```
List Of Modules:
CAT.SCH.TEST.EXE
```

```
1 module found, 0 modules extracted
0 mxcmp invocations: 0 succeeded, 0 failed
```

Note. The OSS tool `mxCompileUserModule` with the `-m` option does not display the module files associated with the DLLs loaded to an executable.

MXCMP Environment Variable

To specify an alternate location of the SQL/MX compiler (MXCMP) instead of the default location of `/G/system/system/mxcmp`, use the `MXCMP` environment variable. This environment variable is used by `ecobol` or `nmcobol` and the `mxCompileUserModule` utility and enables you to direct them to use another version of the MXCMP executable.

To set the `MXCMP` environment variable, enter this command at an OSS prompt before invoking the `ecobol` or `nmcobol` or `mxCompileUserModule` utility:

```
export MXCMP="/G/usr/mydir/mxcmp"
```

For more information, see the *Open System Services Shell and Utilities Reference Manual*.

MXCMPUM Environment Variable

To specify an alternate location of the compiler utility (`mxCompileUserModule`) instead of the default location of `/usr/tandem/sqlmx/bin/mxCompileUserModule`, use the `MXCMPUM` environment variable. This environment variable is used by the `ecobol` or `nmcobol` utility and enables you to direct `ecobol` or `nmcobol` to use another version of the `mxCompileUserModule` utility.

To set the `MXCMPUM` environment variable, enter this command at an OSS prompt before invoking the `ecobol` or `nmcobol` utility:

```
export MXCMPUM="/G/usr/mydir/mxCompileUserModule"
```

For more information, see the *Open System Services Shell and Utilities Reference Manual*.

Compiling a Module Definition File

Embedded SQL application source files preprocessed with the `-x` and `-m` options or that set the `SQLMX_PREPROCESSOR_VERSION=800` environment variable continue to

generate module definition files as done in SQL/MX Release 1.8 and previous releases.

To compile a module definition file, use the SQL/MX compiler `mxcmp` command on the module definition (`.m`) file. The SQL/MX compiler places a compiled user module file in the user-specified local application directory, user-specified Guardian or OSS location(s) or both, application DLL location(s), or in the global `/usr/tandem/sqlmx/USERMODULES` directory.

Command-Line Syntax

To invoke the SQL/MX compiler, at an OSS prompt, enter:

```
mxcmp [ -e ] [ -v ]
      [ -g {moduleGlobal|moduleLocal[=OSSdir]} ]
      [ -d compiler-attribute-name=compiler-attribute-value ] ...
          module-definition-file
```

`-e`

directs `mxcmp` to generate a warning rather than an error if a table or class MAP DEFINE in an SQL statement does not exist during explicit SQL/MX compilation. To find errors in a program during explicit SQL/MX compilation, omit the `-e` option.

If you are using late name resolution and want to use a table or DEFINE that does not exist during explicit SQL/MX compilation, include the `-e` option. Then at run time, the SQL/MX executor automatically recompiles the SQL statement from the statement's source in the module by using the run-time version of the table.

`-v`

directs `mxcmp` to display summary information in addition to error and warning messages for the compilation.

`-g moduleGlobal`

specifies that the module is placed globally in the `/usr/tandem/sqlmx/USERMODULES` directory.

`-g moduleLocal[=OSSdir]`

directs `mxcmp` to place the module in the named OSS directory. The OSS directory can be either a Guardian or an OSS location in the OSS format. If the OSS directory is omitted, the module is created in the current directory. The following rules related to the OSS directory apply:

- The OSS directory must exist and be accessible.
- The OSS directory must not be a remote directory in an Expand network.
- The OSS directory must not exceed 1024 characters.

If these conditions are not met, an error is generated, and no module is created.

If you do not specify `-g moduleLocal=OSSdir`, but set `MXCMP_PLACES_LOCAL_MODULES ON`, you must be in the same directory as the application executable when you invoke `mxcmp`. Otherwise, `mxcmp` writes the module in the current directory, and you will need to move the module to the global `USERMODULES` directory or co-locate the module with its application. For more information, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

`-d compiler-attribute-name=compiler-attribute-value`

specifies default attribute settings for compilation. The default attribute settings for compilation override settings in the `SYSTEM_DEFAULTS` table but do not override the object name qualification or the settings of embedded `CONTROL QUERY DEFAULT`, `DECLARE`, or `SET` statements, which are in the input source file. For more information, see the `SYSTEM_DEFAULTS` table in the *SQL/MX Reference Manual*.

The OSS shell is used to invoke `mxcmp`. Consequently, you must adjust the syntax for setting `CONTROL QUERY DEFAULT` attribute values for `MP_SYSTEM` and `MP_VOLUME`. The OSS shell performs command/parameter substitution and allows a `\` (backslash) to quote special characters such as `$`.

This example shows how to set `MP_SYSTEM` and `MP_VOLUME` as `mxcmp` command-line options:

```
to get MP_SYSTEM=\KINGPIN    --> use -d MP_SYSTEM=\\KINGPIN
to get MP_VOLUME=$TX012      --> use -d MP_VOLUME=\\$TX012
```

You must use a pair of backslashes when specifying the value for `MP_SYSTEM` and one for `MP_VOLUME`.

module-definition-file

is the name of the input module definition file (`.m`) that was generated by the COBOL preprocessor (`mxsqlco`).

The static SQL/MX compiler provides backward compatible behavior. If the `SQLMX_PREPROCESSOR_VERSION` environment variable is set to 800, `mxcmp` behaves just like SQL/MX Release 1.8. Otherwise, `mxcmp` supports all SQL/MX Release 2.x features and command-line options. For more information, see [Influencing Module Management Behavior](#) on page 17-9.

Example—mxcmp

The following command compiles the module definition and places module file in the user specified OSS location, `/usr/mymodules`:

```
mxcmp -g moduleLocal=/usr/mymodules sqlprog.m
```

The following command compiles the module definition and places module file in the user specified Guardian location, /G/data01/mymod:

```
mxcmp -g moduleLocal=/G/data01/mymod sqlprog.m
```

ecobol or nmcobol Utility: Using One Command for All Compilation Steps

In the OSS environment, the `ecobol` or `nmcobol` utility provides the interface to COBOL compilation components, including the SQL/MX COBOL preprocessor, the native COBOL compiler, and the native object file linker (`eld` or `nld`). `ecobol` enables you to perform TNS/E native compilation and build an embedded SQL/MX program in a single command. On systems running H06.05 or later RVUs, `nmcobol` enables you to perform TNS/R native compilation and build an embedded SQL/MX program in a single command. You can also use the compiler utility options individually: for example, to run the SQL/MX compiler after preprocessing.

In the Windows environment, `ecobol` and `nmcobol` are bundled with ETK. For details on syntax and use, see the help file *Using Command-Line Cross Compilers on Windows*, which is included with ETK. In addition, the *Open System Services Shell and Utilities Reference Manual* contains a listing of all `ecobol` and `nmcobol` utility options.

`ecobol` and `nmcobol` are installed in the `/usr/bin` directory.

ecobol and nmcobol Utility Options for SQL/MX

`-Wsqlmx[={ "args" | args }]`

Invokes the SQL/MX preprocessor prior to invoking the COBOL compiler. Cannot be specified with `-Wsql` or `-Wsqlcomp`.

You can use one or more of the *args* (`ansi_format`, `double_quotes`, `listing`, `preprocess_only`, `tandem_format`, or `refrain_r2`), separated by commas without space between them.

`ansi_format`

Directs the SQL/MX preprocessor to assume ANSI fixed format for the source program.

`double_quotes`

Directs the SQL/MX preprocessor to accept SQL string literals delimited by double quotes, in addition to single quotes.

`listing`

Directs the SQL/MX preprocessor to write its diagnostic messages to a file named *file.eL* (where *file* is the name of the primary source file).

`preprocess_only`

Runs the SQL/MX preprocessor only.

`tandem_format`

Directs the SQL/MX preprocessor to assume TANDEM free format for the source program.

`refrain_r2`

Directs the SQL/MX preprocessor to refrain from embedding module definitions in the annotated source file and to use a module definition file.

`-Wsqlmxadd=["args" | arg]`

Passes valid preprocessor commands (*args*) through to the SQL/MX preprocessor without change or validation. The preprocessor validates the syntax.

`-Wsettog=n`

Specifies a numeric toggle from 1 through 15 that is defined only during preprocessing. Ignored without warning if `-Wsqlmx` is not specified. Can be set more than once to set multiple toggles as either:

- `-Wsettog=n,nn...`
separated by commas without any space between them
- `-Wsettog=n -Wsettog=nn...`
separated by space without any commas

The option is not passed to the COBOL compiler. For details about the `-d toggle` option, see [Syntax for the OSS-Hosted SQL/MX COBOL Preprocessor](#) on page 16-14.

`-Wtimestamp=value`

Passes a creation timestamp to the SQL/MX preprocessor. Ignored without warning if `-Wsqlmx` is not specified. If set more than once, only the last occurrence takes effect. `ecobol` or `nmcobol` does not validate *value*. Validation is left to the preprocessor. For details about the form of *value*, see [-t timestamp](#) on page 16-15.

`-Wmxcmp[={ "args" | args }]`

Invokes the SQL/MX compiler. If compiling with embedded module definitions, invokes `mxCompileUserModule`. If compiling with separate module definition files, invokes `mxcmp`. Cannot be specified with `-Wsql`, `-Wsqlcomp`, or `-Wmigrate`. You can use either or both *warn* or *verbose args*, separated by commas without space between them.

warn	Directs the SQL/MX compiler to return a warning rather than an error if a table does not exist at compile time.
verbose	Directs the SQL/MX compiler to display summary information, in addition to error and warning messages.
<code>-Wmxcmp_querydefault= compiler-attribute-name= compiler-attribute-value[, compiler-attribute-value...]</code>	Directs the SQL/MX compiler to issue the control query default setting at the command line. The command-line attribute settings override corresponding entries in the SYSTEM_DEFAULTS table. You can specify multiple attribute name and value pairs, separated by commas without spaces.
<code>-Wmxcmp_add=["args" arg]</code>	Passes any valid set of <code>mxcmp</code> or <code>mxCompileUserModule</code> options (<i>args</i>) to the SQL/MX compiler (<code>mxcmp</code> or <code>mxCompileUserModule</code>) without change or validation. The SQL/MX compiler validates the syntax. You can specify multiple options and value pairs, separated by spaces.
<code>-Wmxcmp_files=args</code>	Passes the <code>.m</code> files specified here to <code>mxcmp</code> for module compilation (with module definition files). Passes all files without the <code>.m</code> file extension to <code>mxCompileUserModule</code> for module compilation (with embedded module definitions in the annotated source file).
<code>-WmoduleCatalog=arg</code>	Directs the SQL/MX preprocessor to use the catalog name if the input <i>sql-file</i> does not have a MODULE directive or its MODULE directive does not specify a catalog name.
<code>-WmoduleSchema=arg</code>	Directs the SQL/MX preprocessor to use the schema name if the input <i>sql-file</i> does not have a MODULE directive or its MODULE directive does not specify a schema name.

<code>-WmoduleGroup=[<i>string</i>]</code>	Directs the SQL/MX preprocessor to group all of an application's module files. The <code>moduleGroup</code> is embedded in the module files' names and enables the use of OSS wild-card file specification patterns to manage the files. For more information, see Grouping on page 17-23.
<code>-WmoduleTableSet=[<i>string</i>]</code>	Directs the SQL/MX preprocessor to use the module management targeting feature. Create different sets of module files that can be used against different sets of tables. For more information, see Specifying the search locations of the module files on page 17-13.
<code>-WmoduleVersion=[<i>string</i>]</code>	Allows multiple versions of an application's module files to coexist while keeping the same <code>MODULE</code> directive in each version. For more information, see Versioning on page 17-21.
<code>-Wsqlmx_pp_defscript=<i>args</i></code>	Specifies the files that contain the class MAP DEFINES that create environment variables before SQL/MX preprocessing.
<code>-Wmxcmp_cmd="oss_command; oss_command"</code>	Specifies the list of OSS commands to execute before invoking the remote <code>mxcmp</code> .
<code>-Wsqlhost={<i>hostname</i> <i>IP-address</i>}</code>	Specifies the host name or IP address of the NonStop system where the tables specified by <code>INVOKE</code> reside. This option is required if you use <code>INVOKE</code> .
<code>-Wsqlloc=<i>OSS-directory</i></code>	Specifies the directory in which module definition files are placed.
<code>-Wsqlmx_port=<i>port-number</i></code>	Specifies the TCP/IP port of the NonStop system to connect to for the ODBC listener process. The default port for the Association server is 18650.
<code>-Wsqluser=<i>user</i>[,<i>password</i>]</code>	Specifies the Guardian user name and password with access to the tables that <code>INVOKE</code> reads. Required if you use <code>INVOKE</code> .

The PC-only options are shaded in gray.

In addition to the options for preprocessing and compiling SQL/MX components, `ecobol` and `nmcobol` supply SQL/MX environment variables that provide the path names for the SQL/MX preprocessor (MXSQLCO) and the SQL/MX compiler (MXCMP and MXCMPUM), in addition to the path name for definitions of the SQL call-level interface (SQLCLIO). These CLI procedure calls, for the translated SQL statements in the annotated COBOL source file, are required by the COBOL compiler. For more information on `ecobol` and `nmcobol` environment variables, see the *Open System Services Shell and Utilities Reference Manual*.

-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`, `c99`, and `ecobol`.

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/MX, 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/MX compilations use only SSH.)

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.

SQL/MX Preprocessing

Use the `-wsqlmx[=args]` command to invoke the SQL/MX preprocessor. For a full description of how the OSS-hosted SQL/MX preprocessor works, see [OSS-Hosted SQL/MX COBOL Preprocessor](#) on page 16-13.

Compiling COBOL Statements

Use the `ecobol` utility to compile COBOL statements in a preprocessed file to create a TNS/E native object file. On systems running H06.05 or later RVUs, use the `nmcobol` utility to compile COBOL statements in a preprocessed file to create a TNS/R native object file.

Your application program can run in the OSS or Guardian environment. Use the `ecobol` or `nmcobol -wsystype=oss` option (which is the default) if you want your application to be an OSS program. Use the `ecobol` or `nmcobol -wsystype=guardian` option if you want your application to be a Guardian program. For more information, see [Building SQL/MX Guardian Applications in the OSS Environment](#) on page 16-50.

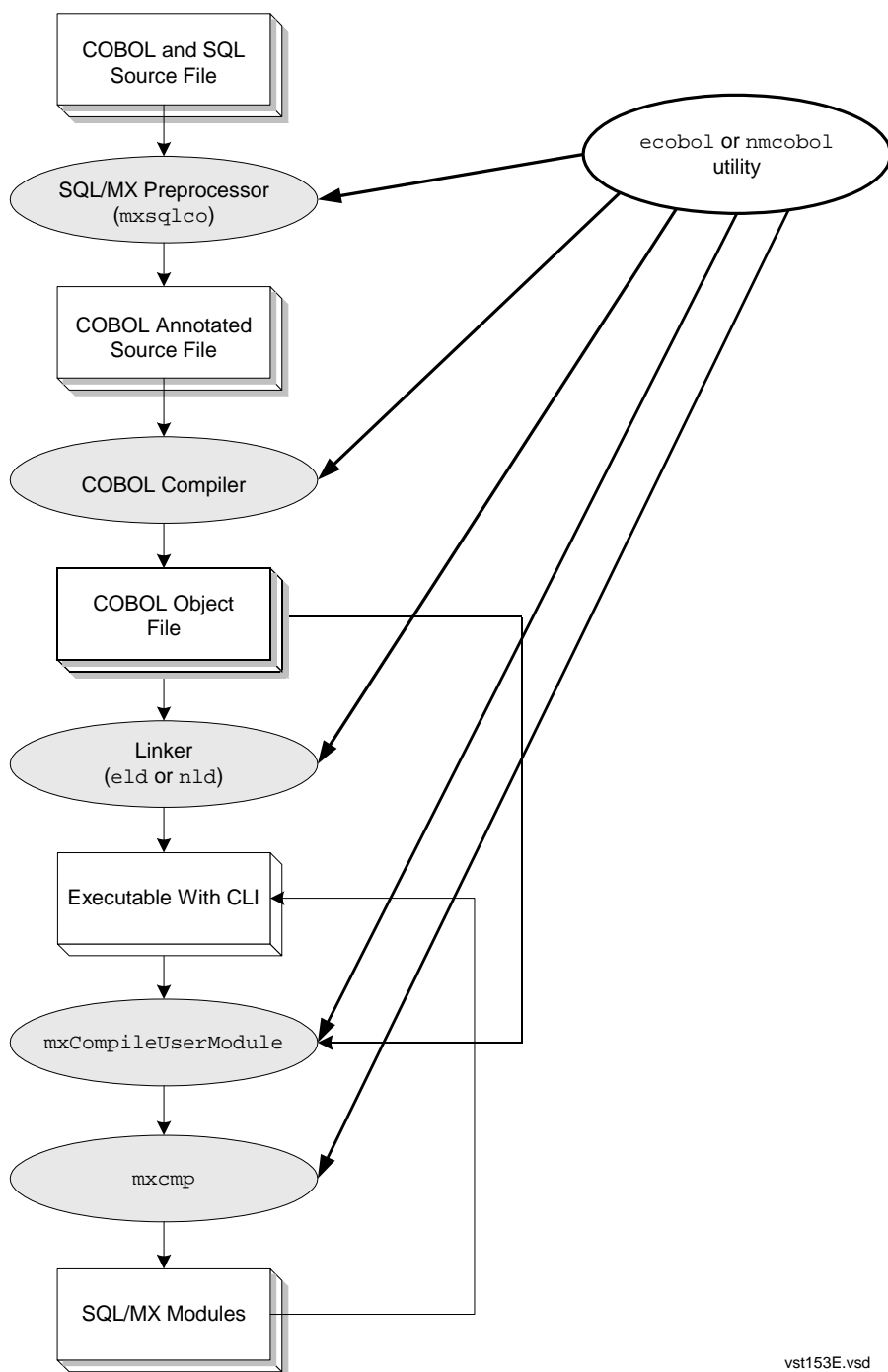
SQL/MX Compiling

Use the `-Wmxcmp [=args]` command to invoke the SQL/MX compiler. For a full description of how to use the SQL/MX compiler, see [Running the SQL/MX Compiler](#) on page 16-25.

ecobol and nmcobol Examples With Embedded Module Definitions

Figure 16-3 shows how the `ecobol` or `nmcobol` utility compiles a COBOL program with embedded module definitions.

Figure 16-3. ecobol or nmcobol Generating Annotated Source With Embedded Module Definitions



vst153E.vsd

- These commands preprocess, compile, link, and SQL compile a single COBOL source file:

```
ecobol -Wsqlmx -Wmxcmp -o sqlprog.exe sqlprog.ecob
nmcobol -Wsqlmx -Wmxcmp -o sqlprog.exe sqlprog.ecob
```

The `ecobol` or `nmcobol` utility invokes the preprocessor, `mxsqlco`, which uses the file `sqlprog.ecob` as input and produces one file: `sqlprog.cbl`, which is a COBOL annotated source file that contains embedded module definitions. The `ecobol` or `nmcobol` utility then compiles and links `sqlprog.cbl` to produce the executable file, `sqlprog.exe`. The SQL/MX compiler command `-Wmxcmp` processes the executable file with the SQL/MX compiler, `mxCompileUserModule`, to produce the module.

- These commands preprocess several COBOL source files and compile them but do not link the results:

```
ecobol -c -Wsqlmx sqlprog1.ecob sqlprog2.ecob \
sqlprog3.ecob
nmcobol -c -Wsqlmx sqlprog1.ecob sqlprog2.ecob \
sqlprog3.ecob
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `sqlprog1.cob`, `sqlprog2.cob`, `sqlprog3.cob`, `sqlprog1.o`, `sqlprog2.o`, and `sqlprog3.o`.

- These commands preprocess and compile COBOL source files with and without embedded SQL without linking:

```
ecobol -c -Wsqlmx file1.cbl file2.ecbl file3.ecob file4.cob
nmcobol -c -Wsqlmx file1.cbl file2.ecbl file3.ecob file4.cob
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `file2.cbl`, `file3.cob`, `file1.o`, `file2.o`, `file3.o`, and `file4.o`.

- The `ecobol` or `nmcobol` utility provides commands that pass through options to the SQL/MX preprocessor and the SQL/MX compiler (`-Wsqlmxadd` and `-Wmxcmp_add`, respectively). To preprocess files with preprocessor options, use the `-Wsqlmxadd` option:

```
-Wsqlmxadd=-a
```

To pass a single option, do not use quotes or white space characters. To pass multiple options, place them within double quotes and separate the options with a white-space character:

```
-Wsqlmxadd="-a -m -c test.cbl"
```

- If you did not link your object files by using the `ecobol` or `nmcobol` utility, create the executable program by using the `eld` or `nld` utility to link one or more object files. For example, these commands link `sqlprog1.o`, `sqlprog2.o`, and find the

required CLI procedure definitions to create an executable file named `sqlprog.exe`:

```
eld -lzcobdll -lzcrcdll -lzclidll sqlprog1.o sqlprog2.o \  
-o sqlprog.exe
```

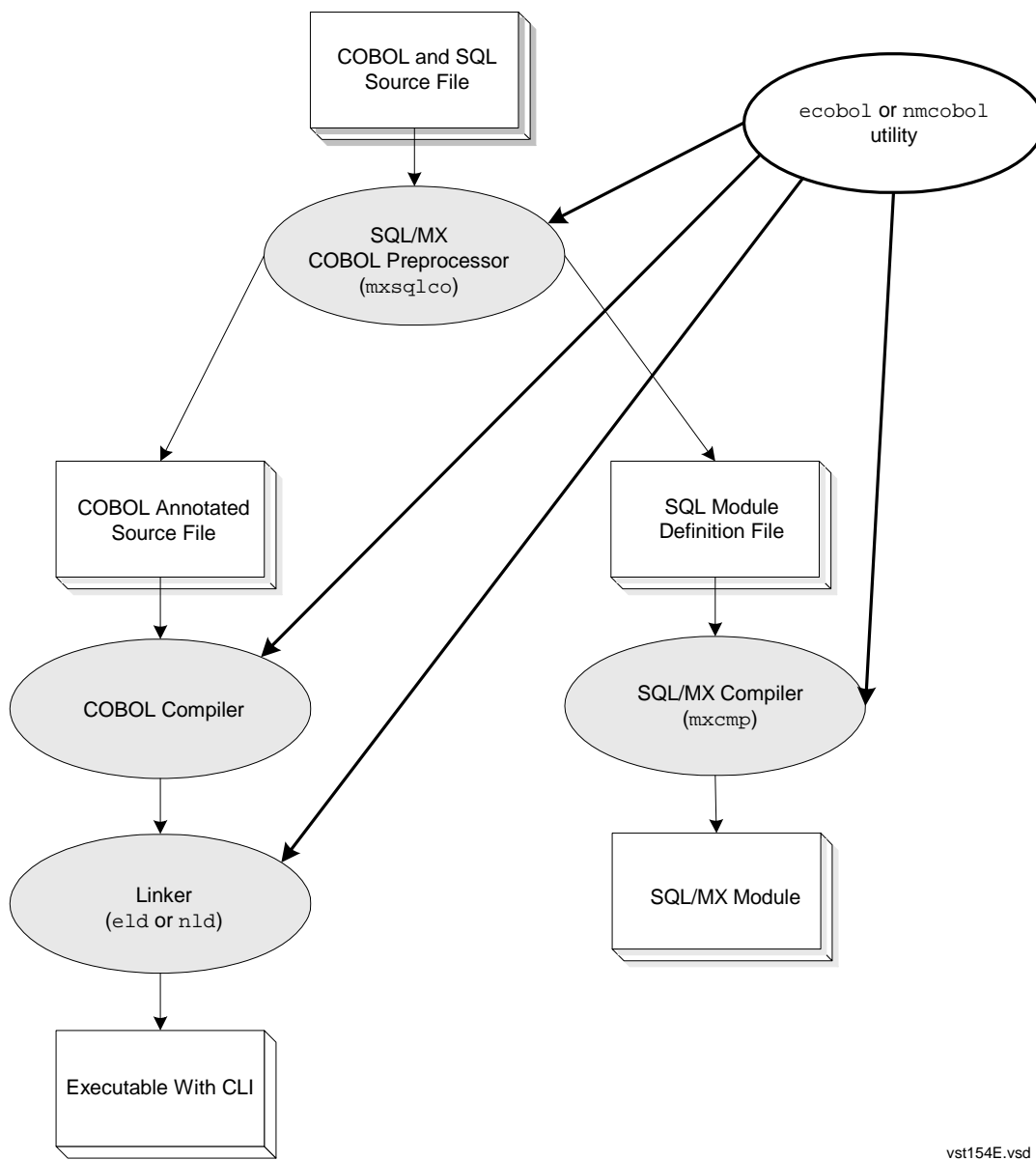
```
nld -lzcobsr1 -lzcresr1 -lzclisr1 sqlprog1.o sqlprog2.o \  
-o sqlprog.exe
```

For more information on `eld`, see the *eld Manual*. For more information on `nld`, see the *nld Manual*.

ecobol and nmcobol Examples With Module Definition Files

[Figure 16-4](#) shows how the `ecobol` or `nmcobol` utility compiles a COBOL program with module definition files.

Figure 16-4. ecobol or nmcobol Generating Module Definition Files



vst154E.vsd

- These commands preprocess, compile, link, and SQL compile a single COBOL source file:

```
ecobol -Wsqlmx -Wsqlmxadd=-x -Wmxcmp -o sqlprog.exe \
sqlprog.ecob sqlprog.m
```

```
nmcobol -Wsqlmx -Wsqlmxadd=-x -Wmxcmp -o sqlprog.exe \
sqlprog.ecob sqlprog.m
```

The `ecobol` or `nmcobol` utility invokes the preprocessor, `mxsqlco`, which uses the file `sqlprog.ecob` as input and produces two files: `sqlprog.cbl` and `sqlprog.m`. The file `sqlprog.cbl` is the COBOL annotated source file, and the file `sqlprog.m` is the corresponding module definition file. The `ecobol` or `nmcobol` utility then compiles and links `sqlprog.cbl` to produce the executable file, `sqlprog.exe`. The SQL/MX compiler command `-Wmxcmp` processes the module definition file `sqlprog.m` with the SQL/MX compiler, `mxcmp`, to produce the module.

- These commands preprocess several COBOL source files and compile them, but they do not link the results:

```
ecobol -c -Wsqlmx -Wsqlmxadd=-x sqlprog1.ecob \
sqlprog2.ecob sqlprog3.ecob
```

```
nmcobol -c -Wsqlmx -Wsqlmxadd=-x sqlprog1.ecob \
sqlprog2.ecob sqlprog3.ecob
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `sqlprog1.m`, `sqlprog2.m`, `sqlprog3.m`, `sqlprog1.cob`, `sqlprog2.cob`, `sqlprog3.cob`, `sqlprog1.o`, `sqlprog2.o`, and `sqlprog3.o`.

- These commands preprocess and compile COBOL source files with and without embedded SQL without linking:

```
ecobol -c -Wsqlmx -Wsqlmxadd=-x file1.cbl file2.ecbl \
file3.ecob file4.cob
```

```
nmcobol -c -Wsqlmx -Wsqlmxadd=-x file1.cbl file2.ecbl \
file3.ecob file4.cob
```

If no errors are detected in either the preprocessing or compiling steps, these files are created: `file2.m`, `file2.cbl`, `file3.m`, `file3.cob`, `file1.o`, `file2.o`, `file3.o`, `file4.o`.

- If you did not link your object files by using the `ecobol` or `nmcobol` utility, create the executable program by using the `eld` or `nld` utility to link one or more object files. For example, these commands link `sqlprog1.o`, `sqlprog2.o`, and find the

required CLI procedure definitions to create an executable file named `sqlprog.exe`:

```
eld -lzcobdll -lzcredll -lzclidll sqlprog1.o sqlprog2.o \
-o sqlprog.exe
```

```
nld -lzcobsrc -lzcresrc -lzclisrc sqlprog1.o sqlprog2.o \
-o sqlprog.exe
```

For more information on `eld`, see the *eld Manual*. For more information on `nld`, see the *nld Manual*.

Combining Embedded Module Definitions and Module Definition Files

Suppose that you have a set of SQL COBOL utility routines that were developed with module definition files. The object code is in `sqlutil.o`. To build, statically link in `sqlutil.o`, and deploy a new application `sqlapp.exe` on OSS:

1. Create the COBOL source file (for example, `sqlapp.ecbl`) that contains embedded SQL/MX statements:

```
* sqlapp.ecbl
?CONSULT sqlutil.o
IDENTIFICATION DIVISION.
PROGRAM-ID. sqlapp.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL DECLARE SCHEMA 'cat.sch' END-EXEC.
    EXEC SQL MODULE sqlappmod END-EXEC.
...
PROCEDURE DIVISION.
...
    EXEC SQL DECLARE get_by_partnum CURSOR FOR
        SELECT partnum, partdesc, price, qty_available
        FROM =parts
        WHERE partnum >= :in_partnum
        FOR UPDATE OF partdesc, price, qty_available;
...

```

2. Set up class MAP DEFINES in the OSS environment by using `add_define`:

```
add_define =parts class=map \
    file=\\bert.\$samdb.sales.parts
```

3. Run the SQL/MX COBOL preprocessor, COBOL compiler, native linker, and SQL/MX compiler in one command:

```
ecobol -Wsqlmx -WmoduleSchema=cobcat.cobsch -Wmxcmp \
-Wmxcmp_querydefault="AUTOMATIC_RECOMPILATION=ON,\
```

```
RECOMPILATION_WARNINGS=ON,SIMILARITY_CHECK=ON" \  
sqlutil.o sqlapp.ecbl -o sqlapp.exe  
  
nmcobol -Wsqlmx -WmoduleSchema=cobcat.cobsch -Wmxcmp \  
-Wmxcmp_querydefault="AUTOMATIC_RECOMPILATION=ON,\  
RECOMPILATION_WARNINGS=ON,SIMILARITY_CHECK=ON" \  
sqlutil.o sqlapp.ecbl -o sqlapp.exe
```

The `ecobol` or `nmcobol` command:

- Preprocesses the COBOL source file, `sqlapp.ecbl`, into an annotated source file, `sqlapp.cbl`, that contains embedded module definitions
- Compiles the annotated source file into an object file, `sqlapp.o`
- Links the object file, `sqlapp.o`, and the SQL COBOL utility routines in `sqlutil.o` into an executable file, `sqlapp.exe`
- Generates a module named `cobcat.cobsch.sqlappmod` in the `USERMODULES` directory

Assuming that the compiled module of `sqlutil.o` is still current, the `sqlapp.exe` executable file is now runnable.

Building SQL/MX COBOL Applications to Run in the Guardian Environment

To build SQL/MX COBOL applications that run in the Guardian environment, choose one of these approaches, depending on your preferred development environment:

- [Building SQL/MX Guardian Applications in the Guardian Environment](#) on page 16-47
- [Building SQL/MX Guardian Applications in the OSS Environment](#) on page 16-50

Building SQL/MX Guardian Applications in the Guardian Environment

- [Using the OSS Pass-Through Command](#) on page 16-47
- [OSS-to-Guardian File Naming](#) on page 16-48
- [Steps for Building an SQL/MX Application in the Guardian Environment](#) on page 16-48
- [Using a TACL Macro to Build an SQL/MX Guardian Application](#) on page 16-49

Using the OSS Pass-Through Command

Most commands for building an SQL/MX Guardian application can be issued directly at a TACL prompt. However, the SQL/MX preprocessor, `mxsqlco`, and the SQL/MX compiler, `mxcmp`, are OSS commands, which run only from OSS. Although `mxcmp`

resides in the Guardian environment, it runs as an OSS process and must be started in the OSS environment.

To run the SQL/MX preprocessor and the SQL/MX compiler in the Guardian environment, use the OSS pass-through command by specifying the `osh -c` option. The `osh -c` option executes one command line at a time in the OSS environment. When you use the `osh -c` command, remember to enclose the entire command string after `osh -c` in double quotes.

Note. When using OSS pass-through commands in the Guardian environment, be aware of the effect of #INFORMAT TACL on those commands. If #INFORMAT TACL is in effect for your session, you must put a tilde (~) before the pipe (|) symbol. Otherwise, the pipe symbol cannot reach the shell for execution because it has a programming function within TACL.

OSS-to-Guardian File Naming

When you issue OSS commands from a TACL prompt to preprocess and SQL compile an application, the Guardian file names change automatically. In the Guardian environment, the period is automatically dropped from the file name.

For example, this OSS pass-through command preprocesses a COBOL source file and generates an annotated source file and module definition file in the Guardian environment:

```
TACL> osh -c "mxsqlco prog.ecob -c prog.cbl -m prog.m"
```

The annotated source file and module definition file in \$MYVOL.MYSUBVOL are PROGCB1 and PROGM. Be aware of the Guardian file name limitation of eight characters.

Steps for Building an SQL/MX Application in the Guardian Environment

Use the next commands at a TACL prompt to preprocess, SQL compile, and compile and link an SQL/MX COBOL program.

1. To make the source file in the Guardian environment accessible to an OSS process, enter this command, replacing `myvol.mysubvol` with your default Guardian volume and subvolume:

```
param home /G/myvol/mysubvol
```

The source file named `progecob` in \$MYVOL.MYSUBVOL must be Guardian file code 101.

2. To invoke the SQL/MX preprocessor, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "mxsqlco progecob -c progcb1 &  
-m prog |tee templog"
```

3. To invoke the SQL/MX compiler, which is an OSS process, enter an OSS pass-through command at a TACL prompt:

```
TACL> osh -c "/G/system/system/mxcmp progmn |tee templog"
```

4. Errors generated by the SQL/MX preprocessor or SQL/MX compiler are logged in the OSS file templog. To convert the error log to a Guardian file:

```
TACL> purge proglog
TACL> ctoedit templog,proglog
```

5. Run the Guardian COBOL compiler and linker.

For TNS/E native compilation:

```
== Convert the annotated source file from an OSS text file
== (file code 180) to a Guardian text file (file code 101).
TACL> ctoedit progcb1,progsr1
== Call the ECOBOL compiler to generate the object file.
TACL> ecobol /in progsr1,out progout/progo; &
        consult $system.system.esqlclio
== Call the eld linker to generate an executable file.
TACL> eld -lzcobdll -lzcresdll -lzclidll progo &
        -o progexe
```

For TNS/R native compilation:

```
== Convert the annotated source file from an OSS text file to
== a Guardian text file.
TACL> ctoedit progcb1,progsr1
== Call the NMCOBOL compiler to generate the object file.
TACL> nmcobol /in progsr1,out progout/progo; &
        consult $system.system.sqlclio
== Call the nld linker to generate an executable file.
TACL> nld -lzcobsr1 -lzcresr1 -lzclisr1 progo &
        -o progexe
```

6. Execute the executable:

```
TACL> run progexe
```

Using a TACL Macro to Build an SQL/MX Guardian Application

Use a TACL macro file to combine and execute the commands. Use these sample TACL macros to customize your own script. In the samples, the source file is located in the Guardian environment and named progecob. Remember that the source file must be Guardian file code 101.

For TNS/E native compilation:

```
?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
```

```

osh -c "mxsqlco progecob -c progcb1 -m prog1 &
~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp prog1 ~|tee -a templog"
== Convert OSS text files (file code 180) to Guardian text files
== (file code 101).
sink [#purge proglog]
ctoedit templog,proglog
ctoedit progcb1,progsr1
== Call the ECOBOL compiler to generate the object file.
ecobol /in progsr1,out progout/progo; &
consult $system.system.esqlclio
== Call the eld linker to generate an executable file.
eld -lzcobd11 -lzcrcd11 -lzcld11 progo -o progexe
== Execute the executable.
run progexe

```

For TNS/R native compilation:

```

?tacl macro
param home /G/myvol/mysubvol
== Store terminal information in file templog.
== The source file must be file code 101.
== Call the SQL/MX preprocessor.
osh -c "mxsqlco progecob -c progcb1 -m prog1 &
~|tee templog"
== Call the SQL/MX compiler.
osh -c "/G/system/system/mxcmp prog1 ~|tee -a templog"
== Convert OSS text files to Guardian text files.
sink [#purge proglog]
ctoedit templog,proglog
ctoedit progcb1,progsr1
== Call the NMCOBOL compiler to generate the object file.
nmcobol /in progsr1,out progout/progo; &
consult $system.system.sqlclio
== Call the nld linker to generate an executable file.
nld -lzcobsr1 -lzcrcsr1 -lzcclsr1 progo -o progexe
== Execute the executable.
run progexe

```

Building SQL/MX Guardian Applications in the OSS Environment

You can use the `ecobol` or `nmcobol -wsystype=guardian` option to build an SQL/MX Guardian application in the OSS environment. Follow these steps:

1. Create an embedded SQL/MX COBOL source file (for example, `prog.ecob`) in the OSS environment.

If your source file contains COPY statements, "OSS " must precede the system file name of an OSS directory. Otherwise, the compiler assumes that referenced files are the same type as `-wsystype`, which is Guardian, and returns an error.

```
COPY TEXT-NAME OF "OSS /usr/ossdir/".
```


2. Compile the COBOL source file by using the `-wsystype=guardian` option of the OSS compiler utility:

```
ecobol -Wsqlmx -Wmxcmp -Wsystype=guardian prog.ecob \
-o prog.exe

nmcobol -Wsqlmx -Wmxcmp -Wsystype=guardian prog.ecob \
-o prog.exe
```

3. Copy the executable file, `prog.exe`, from an OSS directory to a Guardian volume and subvolume:

```
cp prog.exe /G/myvol/mysubvol/progexe
```

4. In the Guardian environment, assign file code 800 (for TNS/E native applications) or file code 700 (for TNS/R native applications) to the executable file:

```
TACL> fup alter progexe, code 800
TACL> fup alter progexe, code 700
```

5. Run the executable in the Guardian environment:

```
TACL> run progexe
```

Running an SQL/MX Application

This subsection describes how COBOL application code is correctly linked to the compiled SQL/MX user module. Topics include:

- [Running the SQL/MX Program File](#) on page 16-52
- [Understanding and Avoiding Common Run-Time Errors](#) on page 16-52
- [Displaying Query Execution Plans](#) on page 16-55

As stated in [Running the SQL/MX COBOL Preprocessor](#) on page 16-9, when the preprocessor reads an embedded SQL source file and writes the COBOL annotated source file, it replaces the SQL statements with COBOL code to call the SQL CLI to execute the SQL statement, along with code to handle parameter passing and error processing. At run time, the calls to the CLI pass in a descriptor of the statement, which gives the statement name, the module name, and a module timestamp.

The CLI begins processing each call by checking that it has the associated module in memory. If not, it uses the module name to find the correct module file in the application's base directory. If a co-located module is not found there, it looks for the module file in the `/usr/tandem/sqlmx/USERMODULES` directory. Before it reads in the compiled SQL plans from a module file, the CLI also checks that the module timestamp encoded in the module file matches the module timestamp passed in from the COBOL application.

If the application consists of more than one separately compiled module, when the first statement from the module is executed, the sequence of reading the module file and checking its module timestamp is performed and repeated for each module associated with the application.

Security of the `/usr/tandem/sqlmx/USERMODULES` directory is very important. You should restrict access so that users cannot alter the query plans in the modules or remove modules. For information on securing modules, see the *SQL/MX Release 3.2 Management Manual*.

Running the SQL/MX Program File

An SQL/MX program can run in the OSS or in the Guardian environment. You can use the `GTACL` command to start a Guardian program from OSS. You can use the `osh` command to start an OSS program from a Guardian TACL session.

- From the OSS environment, enter the program file name at the OSS shell prompt. You can also use the OSS `run` command to run the program file by using specific Guardian attributes (for example, a CPU or priority for the process).
- From the Guardian environment, use the TACL `osh` command to run the program. For more information, see [Building SQL/MX COBOL Applications to Run in the Guardian Environment](#) on page 16-47.

For more information on the `run` or `osh` command, see the *Open System Services Shell and Utilities Reference Manual* or the OSS reference pages.

Understanding and Avoiding Common Run-Time Errors

The details of how a COBOL executable is linked with its module or modules are handled by the system and take place in the background. However, by understanding this process and why certain run-time errors occur, you can avoid some common SQL/MX application development issues.

Module File Errors

Error 8809 Unable to open module file

Error 8809 error occurs if module files are deleted from the base directory of the application, the `/usr/tandem/sqlmx/USERMODULES` directory, user-specified Guardian or OSS location(s) or both, or the application DLL location(s).

This error might also occur if the named module file exists but is not readable, or if the required permission to access the volume, sub volume, or the OSS directory is not granted. The owner of the module file must change the permission attributes to ensure that an application can read the module file.

Error 8808 Module file name contains corrupted or invalid data

This error occurs when the timestamp encoded in the module file does not match the timestamp passed from the application to the CLI. These timestamps are initially generated by the preprocessor and are used to ensure that the version of the application is synchronized with the version of the module file. This error can occur if you run the preprocessor on your embedded SQL, compile the annotated COBOL

output file, but fail to SQL compile the module definition file that the preprocessor generates. If the SQL/MX compiler has previously compiled a different instance of the module definition file, a module exists whose name corresponds to the application module but has a mismatched timestamp.

This error can also occur if you make a copy of an application executable, rebuild the application (thus overwriting the original instance of the application's module file), and then execute the first copy of the application.

A common cause of error 8808 is reuse of code. If you have an embedded SQL source named `myutils.ecob`, you might build and link `myutils` with a number of applications. Each build (that is, preprocessing, COBOL-compilation and SQL compilation) of `myutils` results in a new copy of the same module file overwriting an earlier copy. Only the last application built with `myutils.ecob` avoids error 8808.

To avoid error 8808:

- If you want to reuse embedded modules, use either the grouping or versioning attributes described in [Section 17, Program and Module Management](#). Qualifying your module name with a group or version attribute enables the separate builds of a module to coexist.
- Build `myutils.ecob` only once, and then link the resulting `myutils.o` file to each application.

When you need to rebuild `myutils` for each application, you can either edit the `myutils.ecob` source and change the name of the module that you give in the `MODULE` directive, or you can avoid the `MODULE` directive and let the preprocessor generate the module name.

Error 8400 The `CLASS` attribute of the `DEFINE` is not correct.

Error 8400 occurs if the `Define = _MX_MODULE_SEARCH_PATH` class type is not `SEARCH DEFINE`. This variable is used to locate and load the module file. Ensure that `Define = _MX_MODULE_SEARCH_PATH` is specified correctly and restart the embedded SQL program.

Module File Naming

In application development, avoid the use of delimited identifiers that contain dots (.) in the name of a module's catalog and schema and in the module name itself. Delimited identifiers begin and end with double quotation characters (" "). However, quotation characters are removed when NonStop SQL/MX forms the three-part module name. In some cases of delimited identifiers that contain dots, the resulting three-part module name duplicates an unrelated module name, replacing the query execution plans of the other module file. For example, a module named "A.B".C.D (catalog "A.B", schema C, and module name D) creates a module file name of

`/usr/tandem/sqlmx/USERMODULES/A.B.C.D`. A module named A."B.C".D (catalog A, schema "B.C", and module name D) creates an identically named module file. The second file overwrites the first, and the first module's application cannot

execute. For more information on delimited identifiers, see the *SQL/MX Reference Manual*.

Displaying Query Execution Plans

The EXPLAIN function is an SQL/MX extension that generates a result table describing an access plan for a DML statement, otherwise known as a *query execution plan*. Use the EXPLAIN function for a DML statement in a module. For more information, see [Displaying Query Execution Plans](#) on page 15-75.

Program and Module Management

Developing SQL/MX applications requires both a flexible and controlled development environment. You need to be able to move program and module files from development to a test or production environment. You also need to have control over the development environment to effectively manage module files and modifications made to those files. These needs become even more critical when you consider parallel development on multiple versions of a particular SQL/MX application or when you are testing, deploying, or upgrading SQL/MX applications in a geographically distributed environment.

This section describes how to manage the files of C/C++ and COBOL applications:

- [Program Files](#) on page 17-1
- [Managing Program Files](#) on page 17-3
- [Specifying the search locations for the module files](#) on page 17-7
- [Generating Locally or Globally Placed Modules](#) on page 17-3
- [Module Management Behavior](#) on page 17-8
- [Module Management Naming](#) on page 17-9

Note. NonStop SQL/MX does not support the ability to have different compiled modules of the same application accessing NonStop SQL/MP and NonStop SQL/MX.

Program Files

To make your SQL/MX embedded SQL source files, preprocessed files, module definition files, and module files on both systems easier to find, use the naming conventions listed in [Table 17-1](#).

Table 17-1. File Naming Conventions (page 1 of 2)

File	Naming Convention
Embedded SQL source file in C	<i>source-file.sql</i> <i>source-file.ec</i>
Embedded SQL source file in C++	<i>source-file.eC</i> <i>source-file.ecc</i> <i>source-file.ecpp</i> <i>source-file.ecxx</i> <i>source-file.ec++</i>
Embedded SQL source file in COBOL	<i>source-file.ecob</i> <i>source-file.ecbl</i> <i>source-file.ECOB</i> <i>source-file.ECBL</i>
C preprocessed file	<i>source-file.c</i>

Table 17-1. File Naming Conventions (page 2 of 2)

File	Naming Convention
C++ preprocessed file	<i>source-file.C</i> <i>source-file.cc</i> <i>source-file.cpp</i> <i>source-file.cxx</i> <i>source-file.c++</i>
COBOL preprocessed file	<i>source-file.cbl</i>
Executable program file	<i>source-file.exe</i>
Module definition file	<i>source-file.m</i>
SQL/MX module	<p>Use the MODULE directive to name a module. If you do not specify the catalog and schema names, NonStop SQL/MX automatically qualifies the module name with your current default catalog and schema. If no default catalog and schema are defined, NonStop-hosted SQL/MX preprocessors use your Guardian group and user name for the default module catalog and schema names. Windows-hosted SQL/MX preprocessors use SQLMX_DEFAULT_CATALOG and SQLMX_DEFAULT_SCHEMA as the default module catalog and schema names for the MODULE directive.</p> <p>If you do not use the MODULE directive to name a module, the preprocessor generates a default name.</p> <p>The complete name of the module (module file) is displayed by the SQL/MX compiler after a successful compilation. It is also written into the module definition file's module statement, if one is generated. You can examine the contents of the module definition file with any text editor. For information on module names in embedded module definitions, see Compiling Embedded Module Definitions on page 15-37.</p> <p>By default, modules are stored in <code>/usr/tandem/sqlmx/USERMODULES</code>. However, you can use the <code>mxcmp</code> options to place modules locally, at user-specified Guardian or OSS location(s) or both, or at the application DLL location(s). For more information, see Generating Locally or Globally Placed Modules on page 17-3.</p>

Managing Program Files

You probably develop, test, and debug applications on a development or test system and then move the applications to a production system for actual use. On the development system, you would typically test and tune applications by using a database modeled after the database on the production system.

On the production system, program management tasks include:

- Moving programs from development to production
- Distributing programs across nodes
- Ensuring proper name resolution
- Assigning permissions for running database applications
- Maintaining query plan validity
- Backing up and restoring programs

For more information, see the *SQL/MX Release 3.2 Management Manual*. To migrate programs to NonStop SQL/MX Release 2.x, see the *SQL/MX Database and Application Migration Guide*.

Generating Locally or Globally Placed Modules

In NonStop SQL/MX releases prior to Release 2.x, all modules were globally placed modules in the `/usr/tandem/sqlmx/USERMODULES` directory. In SQL/MX Release 2.x, you can place modules globally or locally. A locally placed module resides in an OSS directory other than the `/usr/tandem/sqlmx/USERMODULES` directory and is co-located with its application executable. The format and contents of globally placed modules are identical to locally placed modules.

Generating locally placed modules is easier and provides more flexibility than generating globally placed modules with the grouping, targeting, and versioning options. Instead of generating globally placed modules in the `/usr/tandem/sqlmx/USERMODULES` directory and relying on the grouping, targeting, and versioning attributes to distinguish between the modules of each application, you can co-locate locally placed modules in the same directory as the application.

For compatibility with applications created in prior releases, in SQL/MX Release 2.x the SQL/MX compiler produces a globally placed module unless instructed to produce a locally placed module.

You can specify that locally placed modules always be produced by setting the `MXCMP_PLACES_LOCAL_MODULES` attribute ON in the `SYSTEM_DEFAULTS` table. The system-defined default value for this attribute is OFF, which means that by default all modules are globally placed in the `/usr/tandem/sqlmx/USERMODULES` directory.

On a case-by-case basis, you specify user-defined, locally and globally placed modules with the following command-line options:

- `mxCompileUserModule`
`-g moduleGlobal | -g moduleLocal[=OSSdir]`
- `mxcmp`
`-g moduleGlobal | -g moduleLocal[=OSSdir]`

For additional details about setting the SQL/MX compiler options, see [Compiling Embedded Module Definitions](#) on page 15-37 and [16-25](#) and [Compiling a Module Definition File](#) on page 15-42 and [16-30](#).

Managing the Coexistence of Globally and Locally Placed Modules

While it is recommended that you choose one approach for a production system (locally or globally placed module generation), you might find a need to mix locally and globally placed modules in your development environment. When you create new applications that use locally placed modules but keep existing applications that use globally placed modules, problems might occur if you are not careful.

The SQL/MX executor always searches for the module by first looking locally and then globally. The execution of an application fails or yields unpredictable results if:

- A locally placed module is deleted, and an older or unrelated module of the same name exists in the `/usr/tandem/sqlmx/USERMODULES` directory.
- A non-module file with the same name as a globally placed module exists in the same OSS directory as the executable.

To avoid these problems:

- Use a unique name for each module.
- Use application file names that do not conflict with your carefully chosen module names.

In an environment where globally and locally placed modules coexist, use one of these methods to generate modules.

- [System-Wide Setting for Locally Placed Modules](#) on page 17-4
- [System-Wide Setting for Globally Placed Modules](#) on page 17-5

System-Wide Setting for Locally Placed Modules

To specify that modules are always placed in a local directory, insert the `MXCMP_PLACES_LOCAL_MODULES` attribute and set it to `ON` in the `SYSTEM_DEFAULTS` table. Remember that the setting in the `SYSTEM_DEFAULTS` table affects all users on the same NonStop system.

To co-locate your modules with your application, ensure that you are in the same directory as the application executable when you invoke `mxcmp` or `mxCompileUserModule`. Otherwise, `mxCompileUserModule` and `mxcmp` place the

module in the current directory, and you will need to manually move the module to co-locate it with its application.

To generate globally placed modules in the `/usr/tandem/sqlmx/USERMODULES` directory on a case-by-case basis, use the `-g moduleGlobal` option, as shown next:

```
mxCompileUserModule -g moduleGlobal app.exe
```

For more information, see [Compiling Embedded Module Definitions](#) on page 15-37 and [16-25](#).

System-Wide Setting for Globally Placed Modules

By default, the `mxCompileUserModule` and `mxcmp` commands automatically place globally placed modules in the `/usr/tandem/sqlmx/USERMODULES` directory.

To generate locally placed modules on a case-by-case basis with `mxCompileUserModule`, use the `-g moduleLocal` option, as shown next:

```
mxCompileUserModule -g moduleLocal dir/app.exe
```

To generate locally placed modules on a case-by-case basis with `mxcmp`, use the `-g moduleLocal=OSSdir` option, replacing `OSSdir` with the name of the application directory:

```
mxcmp -g moduleLocal=OSSdir sqlprog.m
```

For more information, see [Compiling Embedded Module Definitions](#) on page 15-37 and [16-25](#) and [Compiling a Module Definition File](#) on page 15-42 and [16-30](#).

Considerations for Co-Locating Locally Placed Modules

Two methods exist for creating locally placed modules. You can set the `MXCMP_PLACES_LOCAL_MODULES` attribute ON in the `SYSTEM_DEFAULTS` table to specify that locally placed modules always be created. You can generate locally placed modules on a case-by-case basis with the `-g moduleLocal` options for `mxcmp` and `mxCompileUserModule`. Consider these issues when co-locating locally placed modules:

- Invoking `mxCompileUserModule dir/app.exe` is not the same as invoking `mxCompileUserModule -g moduleLocal dir/app.exe`.

The directory path in `mxCompileUserModule dir/app.exe` merely states where the application is located. If you have not set the `MXCMP_PLACES_LOCAL_MODULES` attribute ON in the `SYSTEM_DEFAULTS` table, the module is globally placed in the `/usr/tandem/sqlmx/USERMODULES` directory. If you have set the `MXCMP_PLACES_LOCAL_MODULES` attribute ON in the `SYSTEM_DEFAULTS` table, the module is placed in the current directory where `mxcmp` or `mxCompileUserModule` is invoked. To co-locate the module with the application, ensure that you are in the proper directory before you invoke `mxcmp` or `mxCompileUserModule`.

The directory path in `mxCompileUserModule -g moduleLocal dir/app.exe` states both where the application is located and where to co-locate the module with the application. You can invoke `mxcmp` or `mxCompileUserModule` with the `-g moduleLocal` option from any OSS directory and automatically co-locate the module with the application.

- Set the `MXCMP_PLACES_LOCAL_MODULES` attribute only in the `SYSTEM_DEFAULTS` table.

The `MXCMP_PLACES_LOCAL_MODULES` attribute in the `SYSTEM_DEFAULTS` table establishes a system-wide policy. Avoid these ways of setting `MXCMP_PLACES_LOCAL_MODULES`:

- In an embedded `CONTROL QUERY DEFAULT` statement in an application
- With the `-d` option in `mxcmp` or `mxCompileUserModule`

The syntax for these methods is valid but could be complicated and problematic. For example, this command places the module in the current directory where you invoke `mxCompileUserModule` instead of in the same directory as the application executable:

```
mxCompileUserModule -d MXCMP_PLACES_LOCAL_MODULES=ON
dir/app.exe
```

Instead, you can more easily invoke this command from any OSS directory to co-locate the module with the application:

```
mxCompileUserModule -g moduleLocal dir/app.exe
```

Embedding a static `CONTROL QUERY DEFAULT` statement with `MXCMP_PLACES_LOCAL_MODULES 'ON'` directs the SQL/MX compiler to generate a locally placed module for the application in the OSS directory where you invoke `mxCompileUserModule` or `mxcmp`. Embedding a dynamic `CONTROL QUERY DEFAULT` statement with `MXCMP_PLACES_LOCAL_MODULES 'ON'` directs any SQL/MX compiler process spawned by the application to generate locally placed modules in the current directory.

Therefore, for the best results, set the `MXCMP_PLACES_LOCAL_MODULES` attribute in the `SYSTEM_DEFAULTS` table or set locally or globally placed modules on a case-by-case basis with the `-g moduleLocal` or `-g moduleGlobal` options of `mxcmp` and `mxCompileUserModule`.

Generating modules in a user-specified location

You can generate modules in any user-specified Guardian or OSS location, including local, global, and application DLL locations.

Depending on the requirement, you can specify the user-specified, locally, and globally placed modules with the following command-line options:

```
mxCompileUserModule
-g moduleGlobal | -g moduleLocal=[OSSdir]
mxcmp
-g moduleGlobal | -g moduleLocal=[OSSdir]
```

Specifying the search locations for the module files

The modules placed in the user-specified locations are loaded by the embedded SQL program using `Define =_MX_MODULE_SEARCH_PATH` or the OSS environment variable `_MX_MODULE_SEARCH_PATH`.

Guardian DEFINE

The Guardian DEFINE class must be SEARCH. You can specify either a single location or multiple locations using the SUBVOL0-20 or RELSUBVOL0-20 or both attributes of SEARCH DEFINE. Guardian DEFINE is supported for both Guardian and OSS embedded SQL programs. The order of the module file search is SUBVOL0 , RELSUBVOL0...SUBVOL20 , RELSUBVOL120.

The following are examples of Guardian DEFINE:

- Single search location

```
add define =_MX_MODULE_SEARCH_PATH, class search, subvol0
$DATA01.USRMODS
```

- Multiple search locations

```
add define =_MX_MODULE_SEARCH_PATH, class search, subvol0
($DATA01.USRMOD1,$DATA02.USRMOD2)
```

OSS environment

You can specify multiple OSS locations using the OSS environment variable. The OSS locations must be separated by colons (:). Therefore, SQL/MX does not support module files placed in a directory that contains a colon. If you are using the OSS environment variable, and you want to locate the module files in a Guardian location, you can specify the Guardian module location in the OSS format. For example, `/G/data01/mxmods`. The OSS environment is supported only for OSS-embedded SQL programs.

The module files are searched for in the specified order of the directories, in the environment variable. For example,

```
export
_MX_MODULE_SEARCH_PATH=/home/usermodule1:/home/usermodule2:/G/data01/mxmods
```

Module search sequence

SQL/MX will search for the module file in the following locations, in the specified order:

1. The location of the program executable.
2. Locations specified in the OSS environment variable, `_MX_MODULE_SEARCH_PATH`, followed by the locations specified in `Define` `=_MX_MODULE_SEARCH_PATH`.
3. Each of the application DLL locations.
4. The system global module directory, `/usr/tandem/sqlmx/USERMODULES`.

Note. You must ensure that the module file names are unique across all the locations. While searching for the module files, if SQL/MX finds a module that matches the specified name, it stops searching for the module file and might load the wrong module file.

Managing Modules

For an embedded SQL application to run properly, you must maintain the modules of that application in either the `/usr/tandem/sqlmx/USERMODULES` directory or a locally defined directory. For information on the consequences of mismanaged modules, see [Running an SQL/MX Application](#) on page 15-72 (C/C++) or [Running an SQL/MX Application](#) on page 16-51 (COBOL). For information on locally placed modules, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

Module management tasks include:

- Securing the modules
- Checking module dependencies by using the `DISPLAY USE OF` command
- Removing modules

For more information, see the *SQL/MX Release 3.2 Management Manual*.

Module Management Behavior

By default, in SQL/MX Release 2.x, the SQL preprocessor generates a self-contained application executable file that contains embedded module definitions. To obtain the full benefit of the self-contained application code, avoid using the `-x` or `-m` options when preprocessing embedded SQL source files of an application.

If you have applications that use module definition files, you can continue to generate them by using the environment variable `SQLMX_PREPROCESSOR_VERSION=800` and certain preprocessor options, described under [Influencing Module Management Behavior](#). The SQL preprocessor generates a separate module definition file that contains the translated SQL statements of an embedded SQL source file.

Influencing Module Management Behavior

You might have static SQL/MX application build scripts that require that the SQL preprocessor always generate separate module definition files. By using the environment variable `SQLMX_PREPROCESSOR_VERSION=800` and preprocessor options `-x` and `-m`, you can instruct the SQL preprocessor to generate module definition files.

To set the `SQLMX_PREPROCESSOR_VERSION` environment variable, enter this command on the OSS command line:

```
export SQLMX_PREPROCESSOR_VERSION=800
```

The preprocessor interprets the `SQLMX_PREPROCESSOR_VERSION` environment variable and the `-m` and `-x` options, as indicated in [Table 17-2](#). Suppose that the input file name `sql-file` consists of `base.extension`. Consequently, if the input file name `sql-file` were “`prog.sql`,” its *base* would be “`prog`,” and its *extension* would be “`sql`.”

Table 17-2. Preprocessor Interpretation of `SQLMX_PREPROCESSOR_VERSION` Environment Variable and `-m` and `-x` Options

SQLMX_PREPROCESSOR_VERSION environment variable	-m ?	-x ?	Module Definition File?	Embedded Module Definition? *
Is ≥ 1200 or is not set at all	No	No	No	Yes
Is ≥ 1200 or is not set at all	No	Yes	Yes in <i>base.m</i>	No
Is ≥ 1200 or is not set at all See note **	Yes	No	Yes in <i>module-definition-file</i>	Yes
Is ≥ 1200 or is not set at all	Yes	Yes	Yes in <i>base.m</i>	No
Is set to 800	No	Error	Yes in <i>base.m</i>	No
Is set to 800	Yes	Error	Yes in <i>module-definition-file</i>	No

* The embedded module definition is included in the annotated output source file.

** This row shows the settings that you use to generate both module definition file and a single-file annotated output source file (that contains embedded module definitions). You use these settings when you have a combination of NonStop SQL/MX Release 1.x and 2.x.

Module Management Naming

Use module management naming to externally qualify the file names of modules to assist you with these development tasks:

- Configuring applications to target different sets of database objects
- Managing different versions of an application
- Grouping the modules of an application

To accomplish these tasks, you need not change the C, C++, or COBOL source file or rely strictly on the MODULE directive or on module naming defaults. Instead, you can specify the catalog and schema and the target, version, and group attributes for the module names during SQL preprocessing. The external naming of the module files during preprocessing influences the targeting, versioning, and grouping properties of an application.

For more information, see:

- [How Modules Are Named](#) on page 17-10
- [Effect of Module Management Naming](#) on page 17-13
- [Specifying the search locations of the module files](#) on page 17-13
- [Versioning](#) on page 17-21
- [Grouping](#) on page 17-23

How Modules Are Named

The SQL preprocessor generates a three-part module name based on the MODULE directive that you specify in the C, C++, or COBOL source file. For more information, see [Section 15, C/C++ Program Compilation](#) and [Section 16, COBOL Program Compilation](#). The three parts of the module name are the catalog, schema, and module names, separated by periods. For example:

```
CAT . SCH . MOD
```

If the C, C++, or COBOL source file does not contain a MODULE directive, the SQL preprocessor generates a synthetic three-part module name. If a default catalog exists, the preprocessor uses the default catalog name. If no default catalog exists, the Windows-hosted preprocessor uses the synthetic SQLMX_DEFAULT_CATALOG_ name and the NSK-hosted SQL preprocessor uses your Guardian group name. If a default schema exists, the preprocessor uses the default schema name. If no default schema exists, the Windows-hosted preprocessor uses the synthetic SQLMX_DEFAULT_SCHEMA name and the NSK-hosted preprocessor uses your Guardian group name. If no MODULE directive exists in the source file, the SQL preprocessor introduces a synthetic module name of the form SQLMX_DEFAULT_MODULE_*timestamp*.

In this example, a default catalog and schema are in effect, but no module directive exists:

```
CAT . SCH . SQLMX_DEFAULT_MODULE_21194398887224944
```

In this example, no default catalog or schema are in effect, however, a module directive exists:

```
SQLMX_DEFAULT_CATALOG_ . SQLMX_DEFAULT_SCHEMA_ . TESTA193M (on PC)
GROUP . USER . TESTA193M (on NSK)
```


In this example, no default catalog or schema are in effect and no module directive exists.

```
SQLMX_DEFAULT_CATALOG_.SQLMX_DEFAULT_SCHEMA_.SQLMX_DEFAULT_MODULE_2119439730011160670 (on PC)
GROUP.USER.SQLMX_DEFAULT_MODULE_2119439730011160670 (on NSK)
```

In addition to the three-part module name determined by the MODULE directive or by the system, you can pass these optional module management attributes to the SQL preprocessor:

- Catalog
- Schema
- Target or table set
- Version
- Group

You specify these attributes with the preprocessor option `-g` and without changing the MODULE directive in the source file. The preprocessor applies the catalog and schema and the target (or table set), version, and group attributes to the processed, three-part module name to create an externally qualified module name.

The externally qualified module name consists of the three-part module name plus the module management attributes delimited by circumflex (^) characters:

```
CAT.SCH.GRP^MOD^TABLESET^VER
```

The preprocessor embeds the externally qualified module name in the module definition. During SQL compilation, the SQL/MX compiler uses the externally qualified module name in the module definition to name the module file. During run time, the SQL/MX executor uses the externally qualified module name to locate the associated module file and its compiled query execution plan.

Externally qualified module names enable the coexistence of modules that use the same three-part module names, provided that you use a different target or version attribute during preprocessing. For example:

```
CAT.SCH.^MOD^TABLESET1^VERSION1
```

```
CAT.SCH.^MOD^TABLESET2^VERSION1
```

```
CAT.SCH.^MOD^TABLESET2^VERSION2
```

You can also use the group attribute for coexistence, but that is not its main purpose. For more information on the group attribute, see [Grouping](#) on page 17-23.

For information on the exact `-g` syntax for your development environment and application language, see:

- [Syntax for the OSS-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-20
- [Syntax for the Windows-Hosted SQL/MX C/C++ Preprocessor](#) on page 15-28
- [Syntax for the OSS-Hosted SQL/MX COBOL Preprocessor](#) on page 16-14

- [Syntax for the Windows-Hosted SQL/MX COBOL Preprocessor](#) on page 16-19

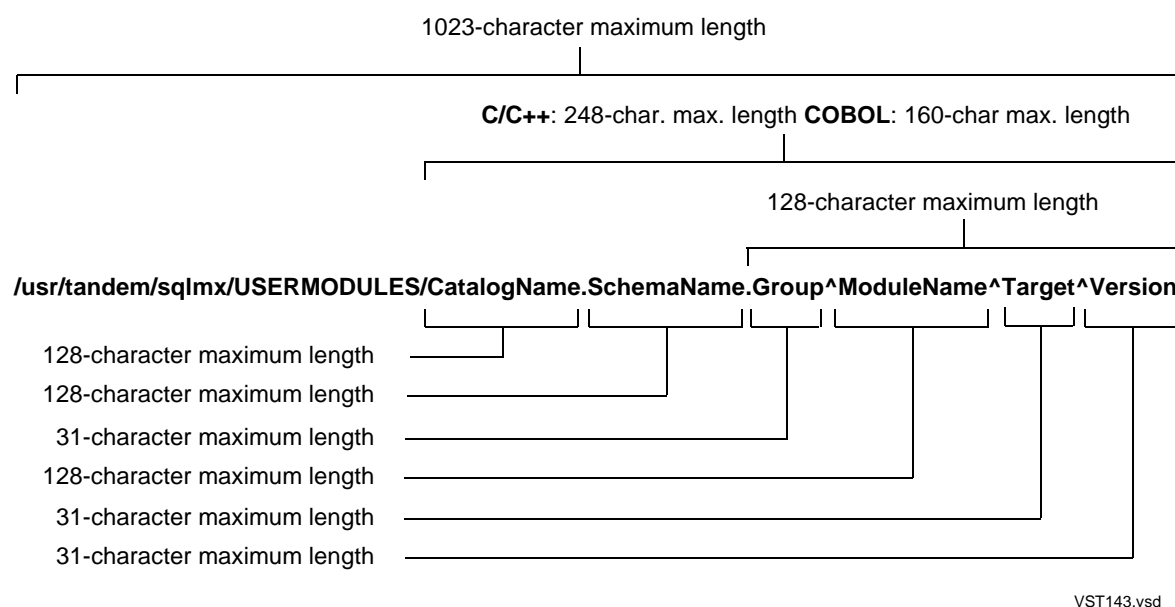
Note. To fully understand the effects of using or not using each of the module management attributes, see [Table 17-3](#) on page 17-13.

Module Name Length

[Figure 17-1](#) shows the limits for module name length. In summary:

- You are not required to choose identifiers that use the maximum length.
- The maximum lengths are not additive.
- The target (or table set), version, and group attributes are optional.

Figure 17-1. Module Name Length



An unqualified module name is limited to maximum 128 characters, as is an externally qualified module name (for example, GRP^MOD^TABLESET^VER). You can choose a catalog name, schema name, or module name each between one character and 128 characters long.

For example, a three-character catalog name plus a three-character schema name plus a 128-character module name plus the two dots (.) between the catalog, schema, and module names constitute a 136-character, three-part module name:

`CAT.SCH.externally-qualified-module-name-that-is-128-characters`

For embedded SQL C/C++ programs, the limit on a fully qualified, three-part module name is 248 characters because this name is eventually used for an OSS file. OSS file names can have a maximum length of 248 characters. For more information, see the *Open System Services User's Guide*.

For embedded SQL COBOL programs, the limit on a fully qualified three-part module name is 160-characters because COBOL restricts nonnumeric literals to at most 160 characters. For more information, see the *HP COBOL85 for NonStop Systems Manual*.

Examples

This example is invalid:

```
cat.sch.grp.modname.target.ver
```

However, this example is valid:

```
cat.sch."grp^modname^target^ver"
```

Note that the third part of the name ("grp^modname^target^ver") is a delimited identifier.

Effect of Module Management Naming

Module management features qualify the name for the module that is encoded in the module definition file and the C-annotated source file. By changing the name of the module in the module definition file, the name of the `module-file` is also changed.

[Table 17-3](#) lists the effects of combinations of group (MGSS), target or table set (MTSS), and version (MVSS) attributes on the module file name.

Table 17-3. Module Management Naming

Group Specified?	TableSet Specified?	Version Specified?	Module Management Qualified Name
Yes	Yes	Yes	CAT.SCH.GRP^MOD^TABLESET^VER
Yes	Yes	No	CAT.SCH.GRP^MOD^TABLESET^
Yes	No	Yes	CAT.SCH.GRP^MOD^^VER
Yes	No	No	CAT.SCH.GRP^MOD^^
No	Yes	Yes	CAT.SCH.^MOD^TABLESET^VER
No	Yes	No	CAT.SCH.^MOD^TABLESET^
No	No	Yes	CAT.SCH.^MOD^^VER
No	No	No	CAT.SCH.MOD

A bold circumflex (^) represents a module management attribute (group, table set, or version) that you did not specify during preprocessing.

Specifying the search locations of the module files

You can specify the user module search locations in one of the following ways:

- Using the Guardian or OSS DEFINE name — This option is supported for both OSS and Guardian SQL/MX applications.
- Using the OSS environment variable — This option is supported only for the OSS SQL/MX applications.

Using the Guardian or OSS DEFINE name

To specify the search locations of the module files using the Guardian or OSS DEFINE name, complete the following steps:

1. Enter the Guardian or OSS DEFINE name. For example, `_MX_MODULE_SEARCH_PATH`.
2. Ensure that the DEFINE class is SEARCH. You can specify the module file locations of SEARCH using `SUBVOL0-20` or `RELSUBVOL0-20` or both attributes.

The following examples illustrate how you can specify the search locations for Guardian Define:

- Single search location

```
add define =_MX_MODULE_SEARCH_PATH, class search, subvol0
$DATA01.USRMODS
```

- Multiple search locations

```
add define =_MX_MODULE_SEARCH_PATH, class search, subvol0
($DATA01.USRMOD1,$DATA02.USRMOD2)
```

The following examples illustrate how you can specify the search locations for OSS Define:

- Single search location

```
add_define =_MX_MODULE_SEARCH_PATH class=SEARCH
SUBVOL0=\\$data04.EDGU
```

- Multiple search locations

```
add_define =_MX_MODULE_SEARCH_PATH class=SEARCH
RELSUBVOL0=\\($data04.ord2m1,$data05.relchk,$data03.ord2m2\
)
```

Using the OSS environment variable

You can use the OSS environment variable, `_MX_MODULE_SEARCH_PATH`, to specify the search locations for the module files.

For example,

```
export _MX_MODULE_SEARCH_PATH=/home/usermodule1:
/home/usermodule2
```

Targeting

By using the target, or table set, attribute for module management, you can create applications that target different sets of database objects (that is, tables, views, and so on) from a single embedded SQL source file without changing the source code, the default catalog or schema, or the MODULE directive. In the embedded SQL source

file, use class MAP DEFINES for database object names and apply compile-time name resolution (or PROTOTYPE host variables and build into your application the logic to set these variables to their proper values at run time) to build applications that target different sets of database objects without changing the source code. For more information, see [DEFINE Names for SQL/MP Objects](#) on page 8-3 and [Compile-Time Name Resolution for SQL/MP Objects](#) on page 8-6.

The target attribute is necessary if you want two or more targeted applications (and their module files) to coexist and run concurrently on the same NonStop system. Without the target attribute, each build of the application writes an identically named module in the `/usr/tandem/sqlmx/USERMODULES` directory, unless you change the MODULE directive in the source code or unless you have instructed the compiler to generate locally placed modules. The target attribute prevents a subsequently built application from overwriting the module file of the previously built application.

Effect of the Target Attribute

The SQL preprocessor checks for the presence of a Module TableSet Specification String (MTSS), which is a regular or delimited identifier that you specify with preprocessor options. For information on how to specify an MTSS, see [Running the SQL/MX C/C++ Preprocessor](#) on page 15-8 and [Running the SQL/MX COBOL Preprocessor](#) on page 16-9. For information on identifiers, see the *SQL/MX Reference Manual*.

Targeting Example for C: Using ModuleTableSet (MTSS)

C

In the next example, a C application that counts employees in various departments is prepared twice to use different sets of tables. One set of tables is for a test environment, and the other set of tables is for a production environment. In the source file `empcnt.sql`, the application's SQL statements are coded with class MAP DEFINES.

```
EXEC SQL DECLARE COUNT_EMP_BY_DEPT CURSOR FOR
SELECT D.DEPT_NUM, COUNT(E.EMP_NUM)
FROM =DEPT AS D, =EMPLOYEE AS E
WHERE D.DEPT_NUM = E.EMP_DEPTNUM GROUP BY D.DEPT_NUM;
```

The application has a MODULE directive:

```
EXEC SQL MODULE CAT.SCH.EMP_CNT_MODULE NAMES ARE ISO88591;
```

To build the application, targeting a set of tables on a test system:

1. Use the OSS `add_define` command to give `=DEPT` and `=EMPLOYEE` the desired values:

```
add_define =DEPT class=MAP file=\\TEST.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP \
                                file=\\TEST.\$DATA.HR1.EMPLOYEE
```

2. Invoke the preprocessor, specifying a *module-tableset-specification-string* with the `-g` option:

```
mxsqlc empcnt.sql -c empcnt.c -m empcnt.m \
-g moduleTableSet=TEST
```

3. The previous preprocessor step produces a pure C file (`empcnt.c`) and a module definition file (`empcnt.m`). The `c89` utility compiles and links the C file, producing an executable `empcnt.exe`, and the SQL/MX compiler compiles `empcnt.m`. Because the *module-tableset-specification-string* is specified as `TEST`, the module file produced by the SQL/MX compiler is:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^
```

To indicate that the executable was built to use the `TEST` compiled module file, the executable is named `empcnt_test.exe`.

If you want to rebuild the application to target a set of production files, you can do so without changing the source file (`empcnt.sql`):

1. Use the OSS `add_define` command to give `=DEPT` and `=EMPLOYEE` the table location to target a different set of files:

```
add_define =DEPT class=MAP file=\\PROD.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP \
file=\\PROD.\$DATA.HR1.EMPLOYEE
```

2. Invoke the preprocessor, specifying a *module-tableset-specification-string* with the `-g` option and a different value:

```
mxsqlc empcnt.sql -c empcnt.c -m empcnt.m \
-g moduleTableSet=PROD
```

3. The previous step again produces a pure C file, `empcnt.c`, and a module definition file named `empcnt.m`.

Note. These two files (`empcnt.c` and `empcnt.m`) overwrite the C and module definition files that were written the first time the application is built. `mxsqlc` does not check for the existence of identically named files before it writes its output files. Although you can specify different output file names to avoid this situation, you might not care that the second C and module definition files overwrite the first build's files because they are easily reproducible. However, if you want to preserve the output of the preprocessor and avoid overwriting files, see [Targeting Example for C: Using Build Subdirectory](#) on page 17-17 or [Targeting Example for COBOL: Using a Build Subdirectory](#) on page 17-20.

The `c89` utility compiles and links the C file, producing an executable, and the SQL/MX compiler compiles `empcnt.m`. The executable is named `empcnt_prod.exe` to denote that it was prepared to use the module files that are targeted to use the production set of tables.

Because a *module-tableset-specification-string* was specified, the module file produced by the SQL/MX compiler is:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^PROD^
```

After building the application for both sets of tables, you can execute, in any order, either compiled executable (`empcnt_test.exe` or `empcnt_prod.exe`) with its compiled SQL.

When `empcnt_test.exe` is run for the first set of tables, you must set up `DEFINES` `=DEPT` to reference `\TEST.$DATA.HR1.DEPT` and `=EMPLOYEE` to reference `\TEST.$DATA.HR1.EMPLOYEE`.

When `empcnt_prod.exe` is run, you must set up `DEFINE`'s `=DEPT` to reference `\PROD.$DATA.HR1.DEPT` and `=EMPLOYEE` to reference `\PROD.$DATA.HR1.EMPLOYEE`.

You can set up `DEFINES` to reference the production environment set of tables, and then use the `empcnt_test.exe` executable (or vice versa). The result of this action is that the compiled SQL plans are read from `empcnt_test.exe` module `/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^`.

These plans refer to the test environment tables, `\TEST.$DATA.HR1.DEPT` and `\TEST.$DATA.HR1.EMPLOYEE`. However, the SQL/MX executor performs late name resolution (see [Late Name Resolution](#) on page 8-6). Because the run-time tables (specified by setting up the `DEFINES` for the production system), are different from the compile-time tables, the executor performs a similarity check. If the two sets of tables are similar, the plan is used. If the two set of tables are dissimilar, the plan is recompiled. The advantage of compile-time name resolution is lost if `DEFINES` are set differently at run time than at compile-time.

Targeting Example for C: Using Build Subdirectory

To avoid the problem where intermediate files and even the executable can be overwritten when rebuilding for a new target, use OSS environment variables. In this C example, the same source file `empcnt.sql` is built twice using an OSS shell script named `empcnt.sh`:

- Set up class `MAP` `DEFINES`. Then set up an OSS environmental variable, `TableSet`, to supply both a *module-tableset-specification-string* and the name of a subdirectory to which the intermediate files (including the before-link object file) and executable can be written:

```
export TableSet=TEST
add_define =DEPT class=MAP file=\\TEST.$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\TEST.$DATA.HR1.EMPLOYEE
```

- Invoke the shell script `empcnt.sh`. The shell script includes the lines:

```
mkdir ./TableSet
mxsqlc empcnt.sql -c $TableSet/empcnt.c \
                  -m $TableSet/empcnt.m \
                  -g moduleTableSet=$TableSet
c89 -o ./TableSet/empcnt.o ./TableSet/empcnt.c
nld -set systype oss \
```

```

    -obey /usr/lib/libc.obey \
    /usr/lib/crtmain.o\
    ./TableSet/empcnt.o \
    -l zcplsr1 \
    -l zcrtlsr1 \
    -l zcresr1 \
    -l zcplosr1 \
    -l ztlhgsr1 \
    -l ztlhosr1 \
    -Bdynamic \
    -l zclisr1 \
    -o ./TableSet/empcnt.exe
/G/system/system/mxcmp ./TableSet/empcnt.m

```

The shell script makes a subdirectory that is named from the environment variable `TableSet` (which was set to `TEST` in the previous example). The script is written so that the intermediate C and module definition files are written into that subdirectory by the preprocessor and read from that subdirectory by the C compiler and `mxcmp`, respectively. (See references to `TableSet` on the `mxsqlc`, `c89`, and `mxcmp` command lines.)

The script also uses the environment variable `TableSet` on the `c89` command line so that its output object file, `empcnt.o`, is written to the subdirectory. Similarly, the `nld` command line uses the same environmental variable to read `empcnt.o` from that subdirectory and write `empcnt.exe` to it as well.

When you need to create another copy of the application to target a different set of tables, export a different value for `TableSet` and set different values to the two `DEFINEs`. For example:

```

export TableSet=PROD
add_define =DEPT class=MAP file=\\PROD.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\PROD.\$DATA.HR1.EMPLOYEE

```

Then rerun the previous `empcnt.sh` shell script. The second set of intermediate files and the `.exe` file are written to a different subdirectory. As in the first example, two distinct module files have been created by the time the two builds complete:

```

/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^PROD^

```

The two executables can coexist, each in their own subdirectory:

```

TEST/empcnt.exe
PROD/empcnt.exe

```

Targeting Example for COBOL: Using ModuleTableSet (MTSS)

COBOL

In this example, a COBOL application, which counts employees in various departments, is prepared twice to make use of two different sets of tables. One set of

tables is for a test environment, and the other set of tables is for a production environment. In a source file (`empcnt.ecbl`), the application's SQL statements are coded with class MAP DEFINES:

```
EXEC SQL DECLARE COUNT_EMP_BY_DEPT CURSOR FOR
SELECT D.DEPT_NUM, COUNT(E.EMP_NUM)
FROM =DEPT AS D, =EMPLOYEE AS E
WHERE D.DEPT_NUM = E.EMP_DEPTNUM GROUP BY D.DEPT_NUM END-EXEC.
```

The application has a MODULE directive:

```
EXEC SQL MODULE CAT.SCH.EMP_CNT_MODULE NAMES ARE
                                ISO88591 END-EXEC.
```

To build the application, targeting a set of tables on a test system:

1. Use the OSS `add_define` command to give `=DEPT` and `=EMPLOYEE` the desired values:

```
add_define =DEPT class=MAP file=\\TEST.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\TEST.\$DATA.HR1.EMPLOYEE
```

2. Invoke the preprocessor, specifying a *module-tableset-specification-string* with the `-g` option:

```
mssqlco empcnt.ecbl -c empcnt.cbl -m empcnt.m \
                                -g moduleTableSet=TEST
```

3. The previous preprocessor step produces a pure COBOL file (`empcnt.cbl`) and a module definition file (`empcnt.m`). The `nmcobol` utility compiles and links the COBOL file and produces an executable (`empcnt_test.exe`), and the SQL/MX compiler compiles `empcnt.m`. (The user directed `nmcobol` to name the executable `empcnt_test.exe` to indicate that it was prepared for the TEST table set). Because a *module-tableset-specification-string* is specified, the module file produced by the SQL/MX compiler is:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^
```

If you want to rebuild the application to target a set of production files, you can do so without changing the source file `empcnt.ecbl`:

1. Use the OSS `add_define` command to give `=DEPT` and `=EMPLOYEE` the values to target a different set of files:

```
add_define =DEPT class=MAP file=\\PROD.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\PROD.\$DATA.HR1.EMPLOYEE
```

2. Invoke the preprocessor, specifying a *module-tableset-specification-string* with the `-g` option and a different value:

```
mssqlco empcnt.ecbl -c empcnt.cbl -m empcnt.m -g \
                                moduleTableSet=PROD
```

3. The previous step again produces a pure COBOL file (`empcnt.cbl`) and a module definition file (`empcnt.m`).

Note. These two files (`empcnt.cbl` and `empcnt.m`) overwrite the COBOL and module definition files the first time the application is built because `mxsqlco` does not check for the existence of identically named files before it writes its output files. You can specify different output file names to avoid this situation. You might not care that the second COBOL and module definition files overwrite the first build's files because they are easily reproducible. [Targeting Example for COBOL: Using a Build Subdirectory](#) on page 17-20 outlines a method to avoid overwriting files.

The `nmcobol` utility compiles and links the COBOL file, producing an executable (`empcnt.exe`), and the SQL/MX compiler compiles `empcnt.m`.

Because a *module-tableset-specification-string* is specified, the module file produced by the SQL/MX compiler is:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^PROD^
```

After building the application for both sets of tables, the user can execute either compiled executable and its corresponding compiled SQL.

When the `empcnt_test.exe` is run for the first set of tables, the user must again set up `DEFINEs` `=DEPT` to reference `\TEST.$DATA.HR1.DEPT` and `=EMPLOYEE` to reference `\TEST.$DATA.HR1.EMPLOYEE`.

The other set of tables is targeted if the user runs `empcnt_prod.exe` and sets up `DEFINEs` `=DEPT` to reference `\PROD.$DATA.HR1.DEPT` and `=EMPLOYEE` to reference `\PROD.$DATA.HR1.EMPLOYEE`.

You can set up `DEFINEs` to reference the production environment set of tables and then use the `empcnt_test.exe` executable (or vice versa). The result of this action is that the compiled SQL plans are read from module:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^.
```

These plans refer to the test environment tables, `\TEST.$DATA.HR1.DEPT` and `\TEST.$DATA.HR1.EMPLOYEE`. However, the SQL/MX executor performs late name resolution (see [Late Name Resolution](#) on page 8-6). Because the run-time tables (specified by setting up the `DEFINEs` for the production system) are different from the compile-time tables, the executor performs a similarity check. If the two sets of tables are similar, the plan is used. If the two set of tables are dissimilar, the plan is recompiled. The advantage of compile-time name resolution is lost if `DEFINEs` are set differently at run time than they were set at compile-time.

Targeting Example for COBOL: Using a Build Subdirectory

To avoid the problem where intermediate files and even the executable can be overwritten when rebuilding for a new target, use OSS environment variables. In this COBOL example, the same source file `empcnt.ecbl` is built twice using an OSS shell script named `empcnt.sh`.

- Set up class MAP DEFINES. Then set up an OSS environment variable, `TableSet`, to supply both a *module-tableset-specification-string* and the name of a subdirectory to which the intermediate files (including the "before-link object file) and executable can be written.

```
export TableSet=TEST
add_define =DEPT class=MAP file=\\TEST.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\TEST.\$DATA.HR1.EMPLOYEE
```

- Invoke the shell script `empcnt.sh`. The shell script includes the lines:

```
mkdir ./TableSet
mxsqlco empcnt.ecbl -c TableSet/empcnt.cbl \
                -m TableSet/empcnt.m \
                -g moduleTableSet=TableSet
nmcobol -o ./TableSet/empcnt.exe \
        -Wcobol="CONSULT /usr/tandem/sqlmx/lib/sqlcli.o" \
        -lzclisrl ./TableSet/empcnt.cbl
/G/system/system/mxcmp ./TableSet/empcnt.m
```

The shell script makes a subdirectory that is named from the environment variable `TableSet` (which was set to `TEST` in the previous example). The script is written so that the intermediate COBOL and module definition files are written into that subdirectory by the preprocessor and read from that subdirectory by the COBOL compiler and `mxcmp`, respectively. (See references to `TableSet` on the command lines for `mxsqlco`, `nmcobol`, and `mxcmp`.)

The script also uses the environment variable `TableSet` on the `nmcobol` command line so that its output object file (`empcnt.exe`) is written into the subdirectory.

When you need to create another copy of the application to target a different set of tables, export a different value for `TableSet` and set different values into the two DEFINES. For example:

```
export TableSet=PROD
add_define =DEPT class=MAP file=\\PROD.\$DATA.HR1.DEPT
add_define =EMPLOYEE class=MAP
                                file=\\PROD.\$DATA.HR1.EMPLOYEE
```

Then rerun the previous `empcnt.sh` shell script. The second set of intermediate files and the `.exe` file are written to a different subdirectory. As in the first example, two distinct module files have been created by the time the two builds complete:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^TEST^
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^PROD^
```

Versioning

By using the version attribute for module management, you can create and use multiple versions of an application from a single embedded SQL source file without changing the `MODULE` directive or the catalog or schema. In the embedded SQL

source file, you would change the source code according to the new version but keep the same MODULE directive. Versioning and targeting differ in that versioning typically requires you to make minor changes to the source code of the application.

The version attribute is necessary if you want two or more versions of the application (and its module files) to coexist on the same NonStop system. Without the version attribute, each build of the application writes an identically named module in the `/usr/tandem/sqlmx/USERMODULES` directory, unless you change the MODULE directive in the source code or unless you instruct the SQL/MX compiler to generate locally placed modules. The version attribute prevents a subsequently built application from overwriting the module file of the previously built application.

Versioning Guidelines

You specify versioning by using the C/C++ or COBOL preprocessor option `-g` and setting the `moduleVersion` attribute to a *Module-Version-Specification-String* (MVSS). This MVSS is embedded in the module name, which is used to name the module file. As with the targeting feature, you need to take steps to prevent the second version of the executable file from overwriting the first. However, unlike the targeting feature, you can use versioning without using compile-time name resolution, so you need not set up class MAP DEFINES before running the application.

In addition to allowing multiple versions of an application to exist, when a unique MVSS is specified each time a module file is precompiled, you can also perform live upgrades. Because you can distinguish versions by using the MVSS attribute, processes running from an earlier program file can use SQL plans from an earlier version of a module file (for example, version 1), and a new program file can use SQL plans from a newer version of a module file (for example, version 2).

Versioning Example: C Set Up

C

In this example, an environment variable is set (`$ThisVersion`), and a build script is invoked. The script contains this line to run the preprocessor:

```
mxsqlc empcnt.sql -c $ThisVersion/empcnt.c \
                 -m $ThisVersion/empcnt.m \
                 -g moduleVersion=$ThisVersion
```

Similar to the example that showed a build subdirectory used for targeting (see [Targeting Example for C: Using Build Subdirectory](#) on page 17-17), the parts of the build script that invoke `c89` read `empcnt.c` from and write `empcnt.o` to the `$ThisVersion` subdirectory. Similarly, the line invoking `nld` reads `$ThisVersion/empcnt.o` and writes `$ThisVersion/empcnt.exe` to prevent each version's intermediate and executable files from overwriting each other. This scenario also applies to the next COBOL example.

If the environmental variable is set both for v1 and for v2, after the build script is run for each version, an executable exists in subdirectories v1 and v2. These two module files will coexist:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^^V1
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^^V2
```

Versioning Example: COBOL Set Up

COBOL

In this example, an environment variable `ThisVersion` is set, and a build script is invoked:

```
mxsqlco empcnt.ecbl -c $ThisVersion/empcnt.cbl \
                  -m $ThisVersion/empcnt.m \
                  -g moduleVersion=$ThisVersion
```

If the environmental variable is set, once for v1 and once for v2, after the entire build script finishes twice, an executable exists in two subdirectories v1 and v2. These two module files will coexist:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^^V1
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.^EMP_CNT_MODULE^^V2
```

Grouping

All the module files that are generated globally on a particular NonStop system are stored in the `/usr/tandem/sqlmx/USERMODULES` directory, making it difficult to identify, or group, the modules that are associated with a particular application. By using the group attribute for module management, you can match module files to an application and perform basic file management tasks more easily. For example, you can list, copy, or delete all the modules associated with the given application by using a single OSS command.

You can also use the group attribute to create module subsets and perform multiple DISPLAY USE OF operations in much less time than it would take to run a single DISPLAY USE OF operation on all the modules. For more information about grouping modules to run multiple DISPLAY USE OF operations, see the *SQL/MX Release 3.2 Management Manual*.

Instead of using the group attribute for globally placed module management, you can use the locally placed module features to generate modules in directories other than `USERMODULES`. For more information on locally placed modules, see [Generating Locally or Globally Placed Modules](#) on page 17-3.

Setting Up Grouping

Grouping requires you to:

- Use the C/C++ or COBOL preprocessor option `-g` and specify the same *Module-Group-Specification-String* (MGSS) name for each of the modules that you want to manage as a group.

- Use meaningful names for grouping your module file.

Grouping Example: C INVENTORY modules

C

In this example, a C application is built from two modules: `reports.sql` and `utils.sql`. `mxsqlc` names the C and module definition file output file according to default rules. At this point, `INVENTORY` is used as the group name for all the modules in an inventory application, enabling the module files to be referred to by group name.

```
mxsqlc reports.sql -g moduleGroup=INVENTORY
mxsqlc utils.sql -g moduleGroup=INVENTORY
c89 -o reports.o reports.c
c89 -o utils.o utils.c
nld -set systype oss \
    -obey /usr/lib/libc.obey \
    /usr/lib/crtmain.o \
    reports.o \
    utils.o \
    -l zcplsr1 \
    -l zcrtlsr1 \
    -l zcresr1 \
    -l zcplosr1 \
    -l ztlhgsr1 \
    -l ztlhosr1 \
    -Bdynamic \
    -l zclisr1 \
    -o invrep
/G/system/system/mxcmp reports.m
/G/system/system/mxcmp utils.m
```

After the application is built, these two module files exist:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.INVENTORY^REPORTS^^
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.INVENTORY^UTILS^^
```

Now, all the application's module files can be referred to by using one OSS file name wild-card pattern:

```
ls /usr/tandem/sqlmx/USERMODULES/*.*.INVENTORY^^
rm /usr/tandem/sqlmx/USERMODULES/*.*.INVENTORY^^
```

Grouping Example: COBOL INVENTORY modules

COBOL

In this COBOL example, an application is built from two modules, `reports.ecbl` and `utils.ecbl`. `mxsqlco` names the COBOL and module definition file output according to default rules. At this point, `INVENTORY` is used as the group name for all the modules in an inventory application, enabling the module files to be referred to by group name.

```
mxsqlco reports.ecbl -g moduleGroup=INVENTORY
mxsqlco utils.ecbl -g moduleGroup=INVENTORY
```

`nmcobol` is used to compile the pure COBOL files (`reports.cbl` and `utils.cbl`), and `nld` links the compiled object files. The last step is to build the application using `mxcmp` to compile the module definition files, `reports.m` and `utils.m`.

After the application is built, these two module files exist:

```
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.INVENTORY^REPORTS^^  
/usr/tandem/sqlmx/USERMODULES/CAT.SCH.INVENTORY^UTILS^^
```

Now, all the application's module files can be referred to by using one OSS file name wild-card pattern:

```
ls /usr/tandem/sqlmx/USERMODULES/*.*.INVENTORY^*  
rm /usr/tandem/sqlmx/USERMODULES/*.*.INVENTORY^*
```


A C Sample Programs

This appendix presents the steps shown in figures in previous sections as complete C programs.

Using a Static SQL Cursor

[Example A-1](#) executes the steps shown in [Figure 6-1](#) on page 6-2.

C

Example A-1. Using a Static SQL Cursor (page 1 of 3)

```
/* -----
   Description:      Using a Static SQL Cursor
   Statements:      Static DECLARE CURSOR
                   BEGIN WORK
                   OPEN
                   FETCH
                   Positioned UPDATE
                   CLOSE
                   COMMIT WORK
                   WHENEVER
                   GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF61M NAMES ARE ISO88591;

int main()
{
    char SQLSTATE_OK[6]="00000";
    char SQLSTATE_NODATA[6]="02000";

    EXEC SQL BEGIN DECLARE SECTION;
        char SQLSTATE[6];
        unsigned NUMERIC (4)  hv_partnum;           /* Parts table */
        char                  hv_partdesc[19];
        NUMERIC (8,2)         hv_price;
        NUMERIC (5)           hv_qty_available;

        unsigned NUMERIC (4)  in_partnum;           /* WHERE value */

        long                  hv_num;               /* Statement info */
        long                  i;                     /* Used for condition loop */

        char                  hv_sqlstate[6];        /* Condition info */
        VARCHAR                hv_tabname[129];
        VARCHAR                hv_colname[129];
        VARCHAR                hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;
```

C**Example A-1. Using a Static SQL Cursor (page 2 of 3)**

```

SQLSTATE[5]='\0';
SQLSTATE_OK[5]='\0';
SQLSTATE_NODATA[5]='\0';

printf("\n\nThis example uses a static cursor. \n\n");

EXEC SQL WHENEVER SQLERROR GOTO end_prog;
EXEC SQL DECLARE CATALOG 'samdbcat';
EXEC SQL DECLARE SCHEMA 'sales';

/* Declare the static cursor. */
EXEC SQL DECLARE get_by_partnum CURSOR FOR
  SELECT partnum, partdesc, price, qty_available
  FROM parts
  WHERE partnum >= :in_partnum
  FOR UPDATE OF partdesc, price, qty_available;

/* Initialize the host variable in the WHERE clause. */
printf("Enter lowest part number to be retrieved: ");
scanf("%hu", &in_partnum);

EXEC SQL BEGIN WORK;          /* Begin transaction. */

/* Open the cursor. */
EXEC SQL OPEN get_by_partnum;

/* Fetch the first row of the result table. */
EXEC SQL FETCH get_by_partnum
  INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
  /* If qty_available less than 1000, update qty_available. */
  if ( hv_qty_available < 1000 ) {
    EXEC SQL UPDATE parts
      SET qty_available = qty_available + 100
      WHERE CURRENT OF get_by_partnum;
    printf("\nUpdate of part number: %hu\n", hv_partnum);
  }
  /* Fetch the next row of the result table. */
  EXEC SQL FETCH get_by_partnum
    INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;
} /* end while */

/* Close the cursor. */
EXEC SQL CLOSE get_by_partnum;
/* Commit any changes. */
EXEC SQL COMMIT WORK;

```

c**Example A-1. Using a Static SQL Cursor** (page 3 of 3)

```

end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;

if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
    printf("\nThe program completed successfully. \n\n");
else {
    EXEC SQL GET DIAGNOSTICS
        :hv_num      = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_tabname = TABLE_NAME,
            :hv_colname = COLUMN_NAME,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_msgtxt = MESSAGE_TEXT;
        hv_tabname[128]='\0'; hv_colname[128]='\0';
        hv_sqlstate[5]='\0'; hv_msgtxt[128]='\0';
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE: %s\n", hv_sqlstate);
        printf("Message  : %s\n", hv_msgtxt);
    } /* end for */
} /* end if */

return 0;
} /* end main */

```

Ensuring Data Consistency

[Example A-2](#) executes the steps shown in [Figure 14-1](#) on page 14-1.

Example A-2. Using TMF to Ensure Data Consistency

```

/* -----
   Description:      Using TMF to ensure data consistency
   Statements:      SET TRANSACTION
                   BEGIN WORK
                   Searched UPDATE
                   COMMIT WORK
                   ROLLBACK WORK
                   WHENEVER
----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF91M NAMES ARE ISO88591;

int main()
{
    char SQLSTATE_OK[6]="00000";
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

SQLSTATE[5]='\0';
SQLSTATE_OK[5]='\0';
printf("\n\nThis example begins and ends a transaction. \n\n");
EXEC SQL WHENEVER SQLERROR GOTO end_prog;
EXEC SQL DECLARE CATALOG 'samdbcat';
EXEC SQL DECLARE SCHEMA 'sales';

/* First, set the attributes for the transaction. */
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

EXEC SQL BEGIN WORK;                /* Start a transaction. */
/* Update the database by setting customer credit. */
EXEC SQL UPDATE customer SET CREDIT = 'CR';

end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;
if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    printf("\nThe update is committed. \n\n");
    EXEC SQL COMMIT WORK;            /* Commit the changes. */
}
else {
    printf("The update is rolled back. \n\n");
    printf("SQLSTATE: %s \n\n", SQLSTATE);
    EXEC SQL ROLLBACK WORK;          /* Roll back the changes. */
}
return 0;
}

```

Using Argument Lists in Dynamic SQL

[Example A-3](#) executes the steps shown in [Figure 9-1](#) on page 9-3.

Example A-3. Using Argument Lists in Dynamic SQL (page 1 of 3)

```

/* -----
   Description:      Using Argument Lists
   Statements:      PREPARE
                   EXECUTE USING ARGUMENTS
                   DEALLOCATE PREPARE
                   WHENEVER
                   GET DIAGNOSTICS
   ----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF101M NAMES ARE ISO88591;

int main()
{
    char SQLSTATE_OK[6]="00000";
    char SQLSTATE_NODATA[6]="02000";

    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4)    hv_empnum;    /* Employee table */
    char                    hv_first_name[16];
    char                    hv_last_name[21];
    unsigned NUMERIC (8,2)  hv_salary;
    short                   hv_salary_i;

    unsigned NUMERIC (4)    in_empnum;
    char                    hv_sql_stmt[256];

    long                    hv_num;        /* Statement info */
    long                    i;             /* Used for condition loop */
    char                    hv_sqlstate[6]; /* Condition info */
    VARCHAR                 hv_tabname[129];
    VARCHAR                 hv_colname[129];
    VARCHAR                 hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;

    SQLSTATE[5]='\0';
    SQLSTATE_OK[5]='\0';
    SQLSTATE_NODATA[5]='\0';

    printf("\n\nThis example uses argument lists. \n\n");

    EXEC SQL WHENEVER SQLERROR GOTO end_prog;

```

Example A-3. Using Argument Lists in Dynamic SQL (page 2 of 3)

```

/* Move statement with input variable to statement variable. */
strcpy(hv_sql_stmt, "SELECT empnum, first_name,"
        " last_name, salary"
        " FROM samdbcat.persnl.employee"
        " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)");

/* Prepare the statement. */
EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

/* Initialize the input parameter in the WHERE clause. */
printf("Enter the employee number to be retrieved: ");
scanf("%hu", &in_empnum);

/* Execute the prepared statement using the argument lists. */
EXEC SQL EXECUTE sqlstmt
    USING :in_empnum
    INTO :hv_empnum, :hv_first_name, :hv_last_name,
        :hv_salary INDICATOR :hv_salary_i;

if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    /* Process the output values. */
    printf("\nEmpnum is: %hu", hv_empnum);

    hv_first_name[15]='\0';
    printf("\nFirst name is: %s", hv_first_name);
    hv_last_name[20]='\0';
    printf("\nLast name is: %s", hv_last_name);

    if (hv_salary_i < 0)
        printf("\nSalary is unknown\n\n");
    else
        printf("\nSalary is: %.2f\n\n", hv_salary/100.0);
} else if (strcmp(SQLSTATE, SQLSTATE_NODATA) == 0)
    printf("\nNo row with employee number %hu\n\n", in_empnum);

/* Deallocate the prepared statement. */
EXEC SQL DEALLOCATE PREPARE sqlstmt;

```

Example A-3. Using Argument Lists in Dynamic SQL (page 3 of 3)

```

end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;

if (strcmp(SQLSTATE, SQLSTATE_OK) != 0) {
    EXEC SQL GET DIAGNOSTICS
        :hv_num      = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_tabname = TABLE_NAME,
            :hv_colname = COLUMN_NAME,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_msgtxt  = MESSAGE_TEXT;
        hv_tabname[128]='\0'; hv_colname[128]='\0';
        hv_sqlstate[5]='\0'; hv_msgtxt[128]='\0';
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE: %s\n", hv_sqlstate);
        printf("Message  : %s\n", hv_msgtxt);
    } /* end for */
} /* end if */

return 0;
} /* end main */

```

Using SQL Descriptor Areas in Dynamic SQL

Using SQL Descriptor Areas With DESCRIBE

[Example A-4](#) on page A-8 executes the steps shown in [Figure 10-1](#) on page 10-12.

Example A-4. Using SQL Descriptor Areas With DESCRIBE (page 1 of 4)

```

/* -----
   Description:      Using Descriptor Areas With DESCRIBE
   Statements:      ALLOCATE DESCRIPTOR
                   SELECT
                   PREPARE
                   DESCRIBE
                   SET DESCRIPTOR
                   EXECUTE
                   GET DESCRIPTOR
                   DEALLOCATE PREPARE
                   DEALLOCATE DESCRIPTOR
                   WHENEVER
                   GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF111M NAMES ARE ISO88591;

int main()
{
    char SQLSTATE_OK[6]="00000";

    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4)    hv_empnum;    /* Employee table */
    char                    hv_first_name[16];
    char                    hv_last_name[21];
    unsigned NUMERIC (4)    hv_deptnum;
    unsigned NUMERIC (4)    hv_jobcode;
    short                   hv_jobcode_i;
    unsigned NUMERIC (8,2)  hv_salary;
    short                   hv_salary_i;

    unsigned NUMERIC (4)    in_empnum;
    char                    in_columns[80];
    char                    hv_sql_stmt[256];
    long                    hv_desc_max;
    long                    hv_desc_value;
    VARCHAR                 sqlda_name[129]; /* NAME in SQL */
                                   /* descriptor area*/

    long                    hv_num;        /* Statement info */
    long                    i;             /* Used for condition loop */
    char                    hv_sqlstate[6]; /* Condition info */
    VARCHAR                 hv_tabname[129];
    VARCHAR                 hv_colname[129];
    VARCHAR                 hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;

```

Example A-4. Using SQL Descriptor Areas With DESCRIBE (page 2 of 4)

```

SQLSTATE[5]='\0';
SQLSTATE_OK[5]='\0';

printf("\n\nThis example uses SQL descriptor areas. \n\n");

EXEC SQL WHENEVER SQLERROR GOTO end_prog;

/* Initialize the output variables in the SELECT list. */
printf("Enter columns to be retrieved, separate by commas: \n");
gets(in_columns);

/* Concatenate statement with input and output variables. */
strcpy(hv_sql_stmt, "SELECT ");
strcat(hv_sql_stmt, in_columns);
strcat(hv_sql_stmt, " FROM samdbcat.persnl.employee"
        " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)");

/* Allocate the descriptor area for input parameters. */
hv_desc_max = 1;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :hv_desc_max;

/* Allocate the descriptor area for output values. */
hv_desc_max = 6;
EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :hv_desc_max;

/* Prepare the statement. */
EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

/* Describe the SQL descriptor area for input parameter. */
EXEC SQL DESCRIBE INPUT sqlstmt
    USING SQL DESCRIPTOR 'in_sqlda';

/* Describe the SQL descriptor area for SELECT values. */
EXEC SQL DESCRIBE OUTPUT sqlstmt
    USING SQL DESCRIPTOR 'out_sqlda';

/* Initialize the input parameter in the WHERE clause. */
printf("Enter the employee number to be retrieved: ");
scanf("%hu", &in_empnum);

/* Set the value of the input parameter in */
/* the input SQL descriptor area. */
hv_desc_value = 1;
EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :hv_desc_value
    VARIABLE_DATA = :in_empnum;

/* Execute the prepared statement using */
/* the SQL descriptor areas. */
EXEC SQL EXECUTE sqlstmt
    USING SQL DESCRIPTOR 'in_sqlda'
    INTO SQL DESCRIPTOR 'out_sqlda';

```

Example A-4. Using SQL Descriptor Areas With DESCRIBE (page 3 of 4)

```

/* Get the count of the number of output values. */
EXEC SQL GET DESCRIPTOR 'out_sqlda' :hv_num = COUNT;

/* Get the ith output value in NAME field and save */
/* in the correct host variable by testing on NAME. */
for (i = 1; i <= hv_num; i++) {
    memset(sqlda_name, '\0', sizeof(sqlda_name));
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
        :sqlda_name = NAME;
    sqlda_name[128]='\0';
    if (strncmp(sqlda_name, "EMPNUM", strlen("EMPNUM"))==0) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_empnum = VARIABLE_DATA;
        printf("\nEmpnum is: %hu", hv_empnum);
    }
    else
    if (strncmp(sqlda_name, "FIRST_NAME", strlen("FIRST_NAME"))==0) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_first_name = VARIABLE_DATA;
        hv_first_name[15]='\0';
        printf("\nFirst name is: %s", hv_first_name);
    }
    else
    if (strncmp(sqlda_name, "LAST_NAME", strlen("LAST_NAME"))==0) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_last_name = VARIABLE_DATA;
        hv_last_name[20]='\0';
        printf("\nLast name is: %s", hv_last_name);
    }
    else
    if (strncmp(sqlda_name, "DEPTNUM", strlen("DEPTNUM"))==0) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_deptnum = VARIABLE_DATA;
        printf("\nDeptnum is: %hu", hv_deptnum);
    }
    else
    if (strncmp(sqlda_name, "JOBCODE", strlen("JOBCODE"))==0) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_jobcode = VARIABLE_DATA,
            :hv_jobcode_i = INDICATOR_DATA;
        if (hv_jobcode_i < 0)
            printf("\nJobcode is unknown");
        else
            printf("\nJobcode is: %hu", hv_jobcode);
    }
}

```

Example A-4. Using SQL Descriptor Areas With DESCRIBE (page 4 of 4)

```

else
  if (strncmp(sqlda_name, "SALARY", strlen("SALARY"))==0) {
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
      :hv_salary = VARIABLE_DATA,
      :hv_salary_i = INDICATOR_DATA;
    if (hv_salary_i < 0)
      printf("\nSalary is unknown");
    else
      printf("\nSalary is: %.2f", hv_salary/100.0);
  }
else
  printf("\nSqlda_name is: %s", sqlda_name);
} /* end if */

/* Deallocate the prepared statement and */
/* the SQL descriptor areas. */
EXEC SQL DEALLOCATE PREPARE sqlstmt;
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda';
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';

end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;

if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
  printf("\nThe program completed successfully. \n\n");
else {
  EXEC SQL GET DIAGNOSTICS
    :hv_num = NUMBER;
  for (i = 1; i <= hv_num; i++) {
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv_tabname = TABLE_NAME,
      :hv_colname = COLUMN_NAME,
      :hv_sqlstate = RETURNED_SQLSTATE,
      :hv_msgtxt = MESSAGE_TEXT;
    hv_tabname[128]='\0'; hv_colname[128]='\0';
    hv_sqlstate[5]='\0'; hv_msgtxt[128]='\0';
    printf("Table      : %s\n", hv_tabname);
    printf("Column    : %s\n", hv_colname);
    printf("SQLSTATE: %s\n", hv_sqlstate);
    printf("Message   : %s\n", hv_msgtxt);
  } /* end for */
} /* end if */

return 0;
} /* end main */

```

Using SQL Descriptor Areas Without DESCRIBE

[Example A-5](#) executes the steps shown in [Figure 10-1](#) on page 10-12 but without the DESCRIBE statements and without the GET DESCRIPTOR statement (there are no output parameters). Instead of using DESCRIBE, the values of fields in the descriptor area are set explicitly by using the SET DESCRIPTOR statement.

Example A-5. Using SQL Descriptor Areas Without DESCRIBE (page 1 of 3)

```

/* -----
   Description:      Using Descriptor Areas Without DESCRIBE
   Statements:      ALLOCATE DESCRIPTOR
                   UPDATE
                   PREPARE
                   SET DESCRIPTOR
                   EXECUTE
                   DEALLOCATE PREPARE
                   DEALLOCATE DESCRIPTOR
                   WHENEVER
                   GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF113M NAMES ARE ISO88591;

void sql_error();

int main()
{
    char SQLSTATE_OK[6]="00000";
    char SQLSTATE_NODATA[6]="02000";

    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    unsigned NUMERIC (4)    in_jobcode; /* WHERE values */
    VARCHAR                 in_last_name[21];

    VARCHAR                 hv_sql_stmt[256];

    long                    desc_max;
    long                    desc_value;
    long                    desc_type;
    long                    desc_precision;
    long                    desc_scale;
    long                    desc_length;
    EXEC SQL END DECLARE SECTION;

    SQLSTATE[5]='\0';
    SQLSTATE_OK[5]='\0';

    printf("\n\nThis example uses descriptors, no DESCRIBE.\n\n");

    EXEC SQL WHENEVER SQLERROR CALL sql_error;

```

Example A-5. Using SQL Descriptor Areas Without DESCRIBE (page 2 of 3)

```

/* Copy statement with input variables. */
strcpy(hv_sql_stmt, "UPDATE samdbcat.persnl.employee"
        " SET salary = salary * 1.1"
        " WHERE jobcode = CAST(? AS NUMERIC(4) UNSIGNED)"
        " AND last_name = ?");

/* Allocate the descriptor area for input parameters. */
desc_max=2;
EXEC SQL ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :desc_max;

/* Prepare the statement. */
EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

/* Initialize the input parameters in the WHERE clause. */
printf("Enter the jobcode: ");
scanf("%hu", &in_jobcode);
printf("Enter the last name: ");
scanf("%s", &in_last_name);

/* Set the values for the jobcode input parameter. */
desc_value      = 1;
desc_type       = -502; /* Smallint unsigned */
desc_precision  = 4;
desc_scale      = 0;

EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
        TYPE          = :desc_type,
        PRECISION     = :desc_precision,
        SCALE         = :desc_scale,
        VARIABLE_DATA = :in_jobcode;

/* Set the values for the last name input parameter. */
desc_value      = 2;
desc_type       = 12; /* character varying */
desc_length     = 20;

EXEC SQL SET DESCRIPTOR 'in_sqlda' VALUE :desc_value
        TYPE          = :desc_type,
        LENGTH        = :desc_length,
        VARIABLE_DATA = :in_last_name;

EXEC SQL BEGIN WORK;

/* Execute the prepared statement using */
/* the SQL descriptor areas. */
EXEC SQL EXECUTE sqlstmt
        USING SQL DESCRIPTOR 'in_sqlda';

```

Example A-5. Using SQL Descriptor Areas Without DESCRIBE (page 3 of 3)

```

if (strcmp(SQLSTATE, SQLSTATE_NODATA) == 0)
    printf("\nNo rows with Jobcode %d and Last Name %s.\n",
           in_jobcode, in_last_name);
else if (strcmp(SQLSTATE, SQLSTATE_OK) == 0) {
    printf("\nThe update is committed.\n");
    EXEC SQL COMMIT WORK;          /* Commit the changes */
}
else {
    printf("\nThe update is rolled back.\n");
    EXEC SQL ROLLBACK WORK;       /* Roll back the changes */
}
/* Deallocate the prepared statement and */
/* the SQL descriptor area. */
EXEC SQL DEALLOCATE PREPARE sqlstmt;
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqlda';

return 0;
} /* end main */

void sql_error() {
    EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    long                hv_num;
    long                i;
    char                hv_sqlstate[6];
    long                hv_sqlcode;
    VARCHAR              hv_tabname[129];
    VARCHAR              hv_colname[129];
    VARCHAR              hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL GET DIAGNOSTICS
        :hv_num = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_tabname = TABLE_NAME,
            :hv_colname = COLUMN_NAME,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_sqlcode = SQLCODE,
            :hv_msgtxt = MESSAGE_TEXT;
        hv_tabname[128]='\0'; hv_colname[128]='\0';
        hv_sqlstate[5]='\0'; hv_msgtxt[128]='\0';
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE: %s\n", hv_sqlstate);
        printf("SQLCODE  : %d\n", hv_sqlcode);
        printf("Message  : %s\n", hv_msgtxt);
    }
} /* end sql_error */

```

Using a Dynamic SQL Cursor

Using a Dynamic SQL Cursor

[Example A-6](#) executes the steps shown in [Figure 11-1](#) on page 11-2 and uses host variable argument lists for the FETCH INTO statement.

Example A-6. Using a Dynamic SQL Cursor (page 1 of 3)

```

/* -----
   Description:      Using a Dynamic SQL Cursor
   Statements:      PREPARE
                   Dynamic DECLARE CURSOR
                   OPEN
                   FETCH
                   CLOSE
                   WHENEVER
                   GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <string.h>
EXEC SQL MODULE EXF121M NAMES ARE ISO88591;

int main()
{
    char SQLSTATE_OK[6]="00000";
    char SQLSTATE_NODATA[6]="02000";

    EXEC SQL BEGIN DECLARE SECTION;
        char SQLSTATE[6];
        unsigned NUMERIC (4) hv_partnum;           /* Parts table */
        char                hv_partdesc[19];
        NUMERIC (8,2)        hv_price;
        NUMERIC (5)          hv_qty_available;
        NUMERIC (5)          in_qty_available; /* Input parameter */

        char                curspec[256];          /* Dynamic cursor spec */

        long                hv_num;                /* Statement info */
        long                i;                    /* Used for condition loop */
        char                hv_sqlstate[6];         /* Condition info */
        VARCHAR             hv_tabname[129];
        VARCHAR             hv_colname[129];
        VARCHAR             hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;

```

Example A-6. Using a Dynamic SQL Cursor (page 2 of 3)

```

SQLSTATE[5]='\0';
SQLSTATE_OK[5]='\0';
SQLSTATE_NODATA[5]='\0';

printf("\n\nThis example uses a dynamic cursor. \n\n");

EXEC SQL WHENEVER SQLERROR GOTO end_prog;

strcpy(curspec,"SELECT partnum, partdesc, price, qty_available"
      " FROM samdbcat.sales.parts "
      " WHERE qty_available <= CAST(? AS NUMERIC(5))");

/* Prepare the cursor specification. */
EXEC SQL PREPARE cursor_spec FROM :curspec;

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

/* Initialize the parameter in the WHERE clause. */
printf("Enter the quantity to initiate reorder: ");
scanf("%d", &in_qty_available);

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

/* Fetch the first row of the result table. */
EXEC SQL FETCH get_by_partnum
      INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;

while (strcmp (SQLSTATE, SQLSTATE_NODATA) != 0) {
    printf("\nOrder part number: %hu, Current qty: %d",
          hv_partnum, hv_qty_available);

    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_by_partnum
          INTO :hv_partnum,:hv_partdesc,:hv_price,:hv_qty_available;
}

/* Close the cursor. */
EXEC SQL CLOSE get_by_partnum;

end_prog:
EXEC SQL WHENEVER SQLERROR CONTINUE;

```

Example A-6. Using a Dynamic SQL Cursor (page 3 of 3)

```

if (strcmp(SQLSTATE, SQLSTATE_OK) == 0)
    printf("\nThe program completed successfully. \n\n");
else {
    EXEC SQL GET DIAGNOSTICS
        :hv_num      = NUMBER;
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_tabname = TABLE_NAME,
            :hv_colname = COLUMN_NAME,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_msgtxt = MESSAGE_TEXT;
        hv_tabname[128]='\0'; hv_colname[128]='\0';
        hv_sqlstate[5]='\0'; hv_msgtxt[128]='\0';
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE: %s\n", hv_sqlstate);
        printf("Message  : %s\n", hv_msgtxt);
    } /* end for */
} /* end else */

return 0;
} /* end main */

```

Using a Dynamic SQL Cursor With Descriptor Area

[Example A-7](#) on page A-18 executes the steps shown in [Figure 11-1](#) on page 11-2 but without using the host variable argument lists for the FETCH INTO statement. Instead of argument lists, the values of the output parameters are stored in the descriptor area, retrieved by using the GET DESCRIPTOR statement, and assigned to a compatible host variable by testing on the data type.

You would probably choose this program to enter a general cursor specification of the form: `SELECT * FROM catalog.schema.table.`

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 1 of 8)

```

/*Description:      Using a Dynamic Cursor With Desc Areas
Statements:        ALLOCATE DESCRIPTOR
                   PREPARE
                   DESCRIBE OUTPUT
                   Dynamic DECLARE CURSOR
                   OPEN
                   FETCH USING DESCRIPTOR
                   GET DESCRIPTOR
                   CLOSE
                   DEALLOCATE PREPARE
                   DEALLOCATE DESCRIPTOR
                   WHENEVER
                   GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <string.h>
#include <math.h>

EXEC SQL MODULE SQL12.mysch.t2002s1 NAMES ARE ISO88591;

void run_dynTest(char *chstr);
void assign_to_hv();
void sql_error();
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    long SQLCODE;
EXEC SQL END DECLARE SECTION;

int main()
{
    char teststr[256];

    strcpy(teststr,"select * from testchar");
    run_dynTest(teststr);
    strcpy(teststr,"select * from testint");
    run_dynTest(teststr);
    strcpy(teststr,"select * from testnum");
    run_dynTest(teststr);
    strcpy(teststr,"select * from testpic");
    run_dynTest(teststr);
    strcpy(teststr,"select cdate, ctime,ctimestamp from
        testdatetime;");
    run_dynTest(teststr);

    strcpy(teststr,"select cintervalYM, cintervaldhms from
        testinterval;");
    run_dynTest(teststr);
}

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 2 of 8)

```

void run_dynTest(char *chstr)
{
    char SQLSTATE_OK[6]="00000";
    char SQLSTATE_NODATA[6]="02000";

    EXEC SQL BEGIN DECLARE SECTION;
        VARCHAR                in_curspec[256];
        long                    desc_max;
    EXEC SQL END DECLARE SECTION;

    exec sql declare schema 'SQL12.mysch';
    exec sql set schema 'SQL12.mysch';

    printf("\n\nThis example uses a dynamic cursor with desc area.
    \n\n");

    EXEC SQL WHENEVER SQLERROR CALL sql_error;

    /* Allocate the descriptor area for output parameters. */
    desc_max=100;
    EXEC SQL ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :desc_max;

    /* Input cursor specification. */
    /*
    printf("\nEnter cursor specification (use fully-qualified table
    name):\n");
    gets(in_curspec);
    */
    strcpy(in_curspec,chstr);
    printf("%s\n", in_curspec);

    /* Prepare the cursor specification. */
    EXEC SQL PREPARE cursor_spec FROM :in_curspec;

    EXEC SQL DESCRIBE OUTPUT cursor_spec USING SQL DESCRIPTOR
    'out_sqlda';

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

    /* Open the cursor using the value of the dynamic parameter. */
    EXEC SQL OPEN get_row;

    /* Fetch the first row of the result table. */

    EXEC SQL FETCH get_row
        INTO SQL DESCRIPTOR 'out_sqlda';

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 3 of 8)

```

while (!strcmp (SQLSTATE, "00000") && strcmp(SQLSTATE,"02000")) {
    /* Process values in the fetched row. */
    assign_to_hv();
    /* Fetch the next row of the result table. */
    EXEC SQL FETCH get_row
        INTO SQL DESCRIPTOR 'out_sqlda';
}
/* Close the cursor. */
EXEC SQL CLOSE get_row;

/* Deallocate the prepared statement and the SQLDAs. */
EXEC SQL DEALLOCATE PREPARE cursor_spec;
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqlda';

} /* end run_dynTest */

void assign_to_hv() {
    EXEC SQL BEGIN DECLARE SECTION;
        unsigned short      hv_num;   /* Descriptor fields */
        unsigned short      i;
        VARCHAR              hv_name[129];
        long                 hv_type;
        long                 hv_datetime;
        long                 hv_lead_precision;
        long                 hv_precision;
        long                 hv_scale;
        long                 hv_length;
        long                 hv_indicator;
        char                  hv_char[256]; /* Value variables */
        VARCHAR              hv_varchar[256];
        _int64               hv_longval;
        int                  hv_short;
        float                hv_real;
        double               hv_double;
        date                 hv_date[11];
        time                 hv_time[16];
        timestamp            hv_timestamp[27];
        INTERVAL YEAR TO MONTH hv_y22mo;
        INTERVAL DAY TO SECOND hv_d2s4;
    EXEC SQL END DECLARE SECTION;

    /* Get the count of the number of output values. */
    EXEC SQL GET DESCRIPTOR 'out_sqlda' :hv_num = COUNT;

    /* Get the ith output value and save. */
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_name          = NAME,
            :hv_type          = TYPE_ANSI,
            :hv_datetime      = DATETIME_CODE,
            :hv_lead_precision = LEADING_PRECISION,
            :hv_precision     = PRECISION,
            :hv_scale         = SCALE,

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 4 of 8)

```

        :hv_length      = RETURNED_LENGTH,
        :hv_indicator   = INDICATOR_DATA;
switch (hv_type) {
case 1:
    printf ("Char      : %s, Type: %d, Length: %d, Null: %d",
           hv_name, hv_type, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_char = VARIABLE_DATA;
    hv_char[hv_length] = '\0';
    printf (" Value: %s\n", hv_char);
    break;
case -601:
    printf ("MPvarchar: %s, Type: %d, Length: %d, Null: %d",
           hv_name, hv_type, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_char = VARIABLE_DATA;
    hv_char[hv_length] = '\0';
    printf (" Value: %s\n", hv_char);
    break;
case 2:
    printf ("Numeric: %s, Type: %d, Precision: %d,"
           " Scale: %d, Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    if (hv_scale > 0) {
        hv_double = hv_double/pow(10,hv_scale);
        printf (" Value: %f\n", hv_double);
    } else printf (" Value: %.f\n", hv_double);
    break;
case 3:
    printf ("Decimal: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    if (hv_scale > 0) {
        hv_double = hv_double/pow(10,hv_scale);
        printf (" Value: %f\n", hv_double);
    } else printf (" Value: %.f\n", hv_double);
    break;
case 4:
    printf ("Integer: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    if (hv_scale > 0) {
        hv_double = hv_double/pow(10,hv_scale);

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 5 of 8)

```

    printf (" Value: %f\n", hv_double);
    } else printf (" Value: %.f\n", hv_double);
    break;
case 5:
    printf ("Smallint: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_short = VARIABLE_DATA;
    printf (" Value: %hd\n", hv_short);
    break;
case -411:
    printf ("Float: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_real = VARIABLE_DATA;
    printf (" Value: %f\n", hv_real);
    break;
case -412:
    printf ("Real: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_real = VARIABLE_DATA;
    printf (" Value: %LE\n", hv_real);
    break;
case -413:
    printf ("Double: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    printf (" Value: %LE\n", hv_double);
    break;
case 9:
    switch (hv_datetime) {
    case 1:
        memset(hv_date, ' ', sizeof(hv_date));
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
               :hv_date = VARIABLE_DATA;
        hv_date[hv_length]=0;
        printf ("date Value: %s\n", hv_date);
        break;

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 6 of 8)

```

    case 2:
        memset(hv_time, ' ', sizeof(hv_time));
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_time = VARIABLE_DATA;
        hv_time[hv_length]=0;
        printf ("time Value: %s\n", hv_time);
        break;
    case 3:
        memset(hv_timestamp, ' ', sizeof(hv_timestamp));
        EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv_timestamp = VARIABLE_DATA;
        hv_timestamp[hv_length]=0;
        printf ("timestamp Value: %s\n", hv_timestamp);
        break;
} /* end switch hv_datetime */
break;
case 10:
    switch (hv_datetime) {
        case 7: /* INTERVAL YEAR TO MONTH */
            memset(hv_y22mo, ' ', sizeof(hv_y22mo));
            EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
                :hv_y22mo=VARIABLE_DATA;
            hv_y22mo[hv_length]=0;
            printf ("Interval year to month: %s/n", hv_y22mo);
            break;
        case 10: /* INTERVAL DAY TO FRACTION */
            memset(hv_d2s4, 0, sizeof(hv_d2s4));
            EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
                :hv_d2s4=VARIABLE_DATA;
            hv_d2s4[hv_length]=0;
            printf ("Interval daytosecond: %s", hv_name);
            break;
    } /* end switch hv_datetime */
    break;
case 12:
    printf ("Varchar: %s, Type: %d, Length: %d, Null: %d",
        hv_name, hv_type, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
        :hv_varchar = VARIABLE_DATA;
    printf (" Value: %s\n", hv_varchar);
    break;

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 7 of 8)

```

case -301:
    printf ("Decimal unsigned: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    if (hv_scale > 0) {
        hv_double = hv_double/pow(10,hv_scale);
        printf (" Value: %f\n", hv_double);
    } else printf (" Value: %.f\n", hv_double);
    break;
case -201:
case -401:
    printf ("Integer unsigned: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_double = VARIABLE_DATA;
    if (hv_scale > 0) {
        hv_double = hv_double/pow(10,hv_scale);
        printf (" Value: %f\n", hv_double);
    } else printf (" Value: %.f\n", hv_double);
    break;
case -402:
    printf ("Largeint: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_longval = VARIABLE_DATA;
    printf (" Value: %Ld\n", hv_longval);
    break;
case -502:
    printf ("Smallint unsigned: %s, Type: %d, Precision: %d,"
           " Scale: %d,\n      Length: %d, Null: %d",
           hv_name, hv_type, hv_precision,
           hv_scale, hv_length, hv_indicator);
    EXEC SQL GET DESCRIPTOR 'out_sqlda' VALUE :i
           :hv_short = VARIABLE_DATA;
    printf (" Value: %hu\n", hv_short);
    break;
default:
    printf ("Invalid or unspecified type: %s,"
           "Type: %d, Length: %d, Null: %d",
           hv_name, hv_type, hv_length, hv_indicator);
    break;
} /* end switch hv_type*/
} /* end for */
} /* end assign_to_hv */

```

Example A-7. Using a Dynamic SQL Cursor With Descriptor Areas (page 8 of 8)

```

void sql_error() {
    EXEC SQL BEGIN DECLARE SECTION;
        -- char SQLSTATE[6];
        unsigned short      hv_num;           /* Statement info */
        VARCHAR              hv_cmdfcn[129];
        VARCHAR              hv_dynfcn[129];
        unsigned short      i;               /* Used for condition loop */
        char                 hv_sqlstate[6];  /* Condition info */
        VARCHAR              hv_tabname[129];
        VARCHAR              hv_colname[129];
        VARCHAR              hv_msgtxt[129];
    EXEC SQL END DECLARE SECTION;

    printf("sql_error: SQLSTATE=%s, %ld\n", SQLSTATE, SQLCODE);
    exec sql whenever sqlerror continue;
    EXEC SQL GET DIAGNOSTICS
        :hv_num      = NUMBER,
        :hv_cmdfcn   = COMMAND_FUNCTION,
        :hv_dynfcn   = DYNAMIC_FUNCTION;
    printf("\nStatement: %s %s\n", hv_cmdfcn, hv_dynfcn);
    for (i = 1; i <= hv_num; i++) {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
            :hv_tabname   = TABLE_NAME,
            :hv_colname   = COLUMN_NAME,
            :hv_sqlstate  = RETURNED_SQLSTATE,
            :hv_msgtxt    = MESSAGE_TEXT;
        printf("Table      : %s\n", hv_tabname);
        printf("Column    : %s\n", hv_colname);
        printf("SQLSTATE: %s\n", hv_sqlstate);
        printf("Message  : %s\n", hv_msgtxt);
    }
    exit(1);
} /* end sql_error */

```

Using a Dynamic SQL Rowset

[Example A-8](#) shows a dynamic embedded SQL program that uses descriptor areas.

Example A-8. Dynamic SQL Rowsets (page 1 of 2)

```

/*****
void dynamic_direct()
/*****
{

/* Initialize all variables */
printf("DYNAMIC_DIRECT:\n");
strcpy(in_desc,"inscols ");
SQLSTATE[5] = '\0';
memset(statementBuffer, ' ', 390);
statementBuffer[389] = '\0';
output_rowset_size = 10;

/* INSERTING 10 ROWS */

EXEC SQL DELETE FROM CAT.SCH.DYNAMIC5;

printf("prepare insert:\n");
strcpy(statementBuffer,
"INSERT INTO CAT.SCH.DYNAMIC5 VALUES ( 'jim', ?[10] );" );

/* construct S1 from of INSERT statement */

EXEC SQL PREPARE S1 FROM :statementBuffer;
printf("SQLSTATE after prepare is %s\n", SQLSTATE);
printf("SQLCODE after prepare is %d\n", SQLCODE);
EXEC SQL PREPARE S1 FROM :statementBuffer;

num_in = 30;
/* create SQLDA for INSERT columns */
EXEC SQL ALLOCATE DESCRIPTOR GLOBAL :in_desc with MAX :num_in;
printf("SQLSTATE after allocate is %s\n", SQLSTATE);

/* populate the SQLDA */
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR :in_desc;
printf("SQLSTATE after describe is %s\n", SQLSTATE);

num = 1;

EXEC SQL GET DESCRIPTOR :in_desc :output_rowset_size = ROWSET_SIZE;
printf("ROWSET_SIZE after prepare & describe is %d\n", output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc :output_rowset_size = COUNT;
printf("COUNT after prepare & describe is %d\n", output_rowset_size);

```

Example A-8. Dynamic SQL Rowsets (page 2 of 2)

```

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size =
    ROWSET_VAR_LAYOUT_SIZE;
printf("ROWSET_VAR_LAYOUT_SIZE after prepare & describe is %d\n",
    output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size =
    ROWSET_IND_LAYOUT_SIZE;
printf("ROWSET_IND_LAYOUT_SIZE after prepare & describe is %d\n",
    output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size = TYPE;
printf("TYPE after prepare & describe is %d\n", output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size = TYPE_FS;
printf("TYPE_FS after prepare & describe is %d\n", output_rowset_size);

populateInputHostvars();
arr_ptr = (long) (&(b_arr[0])) ;
ind_ptr = (long) (&(b_arr_ind[0]));
arr_size = 4;
ind_size = 2;

EXEC SQL SET DESCRIPTOR :in_desc VALUE :num
    VARIABLE_POINTER = :arr_ptr,
    INDICATOR_POINTER = :ind_ptr,
    ROWSET_VAR_LAYOUT_SIZE = :arr_size,
    ROWSET_IND_LAYOUT_SIZE = :ind_size;

EXEC SQL GET DESCRIPTOR :in_desc :output_rowset_size = ROWSET_SIZE;
printf("ROWSET_SIZE after prepare & describe is %d\n", output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc :output_rowset_size = COUNT;
printf("COUNT after prepare & describe is %d\n", output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size =
    ROWSET_VAR_LAYOUT_SIZE;
printf("ROWSET_VAR_LAYOUT_SIZE after prepare & describe is %d\n",
    output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size =
    ROWSET_IND_LAYOUT_SIZE;
printf("ROWSET_IND_LAYOUT_SIZE after prepare & describe is %d\n",
    output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size = TYPE;
printf("TYPE after prepare & describe is %d\n", output_rowset_size);

EXEC SQL GET DESCRIPTOR :in_desc VALUE :num :output_rowset_size = TYPE_FS;
printf("TYPE_FS after prepare & describe is %d\n", output_rowset_size);

EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR :in_desc;

EXEC SQL COMMIT ;

printRowsFromDynamic5();

EXEC SQL DELETE FROM CAT.SCH.DYNAMIC5;
EXEC SQL DEALLOCATE DESCRIPTOR :in_desc ;
EXEC SQL DEALLOCATE PREPARE S1;
}

```

Using SQL Descriptors to Select KANJI and KSC5601 Data

DDL for KANJI and KSC4501 Table Columns

[Example A-9](#) creates SQL/MP tables that contain KANJI and KSC4501 columns and inserts data into those columns.

Example A-9. DDL for KANJI and KSC4501 Table Columns (page 1 of 2)

```
CREATE TABLE words( wordInKanji char(20) CHARACTER SET KANJI,
                    wordInKsc5601 char(20) CHARACTER SET KSC5601); -- An
                                                                    SQL/MP
                                                                    table

INSERT INTO words VALUES (_kanji'Japan ', _ksc5601'Japan ');
INSERT INTO words VALUES (_kanji'Korea ', _ksc5601'Korea ');

DROP TABLE international_customer;

CREATE TABLE international_customer (
    custnum          numeric(4) unsigned
                    no default not NULL not droppable,
    custname         CHARACTER(18) CHARACTER SET UCS2
                    no default not NULL not droppable,
    street           CHARACTER(22) CHARACTER SET UCS2
                    no default not NULL not droppable,
    city             CHARACTER(14) CHARACTER SET UCS2
                    no default not NULL not droppable,
    country          CHARACTER(20) CHARACTER SET UCS2
                    no default not NULL not droppable,
    postcode         CHARACTER(10)
                    no default not NULL not droppable,
    primary key      (custnum) not droppable
);

INSERT INTO international_customer VALUES
    (1,
     _UCS2'John Smith',
     _UCS2'102 Main Street',
     _UCS2'Toyko',
     _UCS2'Japan',
     '56789'
    ),
    (2,
     _UCS2'John Smith',
     _UCS2'102 Jefferson Street',
     _UCS2'Shanghai',
     _UCS2'China',
     '23189'
    ),
```

Example A-9. DDL for KANJI and KSC4501 Table Columns (page 2 of 2)

```

                                (3,
                                  _UCS2'Allen Jones',
                                  _UCS2'LA Blvd',
                                  _UCS2'Houston',
                                  _UCS2'USA',
                                  '39189'
                                );

DROP TABLE usa_customer;

CREATE TABLE usa_customer (
  custnum          numeric(4) unsigned
                  no default not NULL not droppable,
  custname         CHARACTER(18)
                  no default not NULL not droppable,
  street           CHARACTER(22)
                  no default not NULL not droppable,
  city             CHARACTER(14)
                  no default not NULL not droppable,
  state            CHARACTER(12)
                  no default not NULL not droppable,
  postcode         CHARACTER(10)
                  no default not NULL not droppable,
  credit           CHARACTER(2) default 'c1'
                  not NULL not droppable,
  primary key      (custnum) not droppable
);

INSERT INTO usa_customer values
                                (1,
                                  'John Smith',
                                  '102 Main Street',
                                  'Houston',
                                  'TX',
                                  '56789',
                                  'C1'
                                ),
                                (2,
                                  'John Smith',
                                  '102 Jefferson Street',
                                  'Cupertino',
                                  'CA',
                                  '23189',
                                  'C1'
                                ),
                                (3,
                                  'Allen Jones',
                                  'LA Blvd',
                                  'Richmond',
                                  'VA',
                                  '39189',
                                  'C1'
                                );

```

Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data

[Example A-10](#) on page A-30 selects KANJI and KSC5601 data from SQL/MP tables by using SQL descriptor areas.

Example A-10. Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data (page 1 of 5)

```

/* -----
Description:  Using SQL Descriptor Areas to pass-in and
              select MP KANJI and KSC5601 data
Statements:  ALLOCATE DESCRIPTOR
              PREPARE
              DESCRIBE
              SET DESCRIPTOR
              EXECUTE
              DEALLOCATE PREPARE
              DEALLOCATE DESCRIPTOR
              WHENEVER
              GET DIAGNOSTICS
----- */

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <wchar.h>

EXEC SQL MODULE MPC SINOUTM NAMES ARE ISO88591;
long SQLCODE;

EXEC SQL
BEGIN DECLARE SECTION;
  int hv_desc_max;
  int i, j;
  char in_sqlda[13];
  char out_sqlda[13];
  char hv_sql_stmt[255];
  long degree;
  long data_type;
  long data_oct_len;
  long data_len;
  long return_len;
  long return_oct_len;
  VARCHAR charset_name[129];

  char CHARACTER SET KANJI hv_input_in_KANJI[21];
  char CHARACTER SET KSC5601 hv_input_in_KSC5601[21];

  VARCHAR CHARACTER SET KANJI hv_output_in_KANJI[21];
  VARCHAR CHARACTER SET KSC5601 hv_output_in_KSC5601[21];

EXEC SQL END DECLARE SECTION;

void handle_error()
{
  EXEC SQL BEGIN DECLARE SECTION;
    long i,num, hv_cond_num,hv_sqlcode;
    char hv_sqlstate[6], hv_table_name[129],hv_column_name[129];
    char hv_message_text[256];
  EXEC SQL END DECLARE SECTION;

```

Example A-10. Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data (page 2 of 5)

```

EXEC SQL WHENEVER SQLERROR GOTO errexit;
exec sql get diagnostics :num = NUMBER;

for (i=1;i<=num;i++) {
    EXEC SQL get diagnostics exception :i
        :hv_cond_num = CONDITION_NUMBER,
        :hv_sqlstate = RETURNED_SQLSTATE,
        :hv_table_name = TABLE_NAME,
        :hv_column_name = COLUMN_NAME,
        :hv_message_text = MESSAGE_TEXT,
        :hv_sqlcode = SQLCODE;
    hv_sqlstate[5] = 0;
    printf("condition number: %d\n", hv_cond_num);
    printf("sqlstate: %s\n", hv_sqlstate);
    printf("table name: %s\n", hv_table_name);
    printf("column name: %s\n", hv_column_name);
    printf("message text: %s\n", hv_message_text);
    printf("sqlcode: %ld\n", hv_sqlcode);
    printf("\n");
}

return;

errexit:
    printf("\nError in the error handler.  SQLCODE = %d\n",
SQLCODE);
    exit(1);
}

EXEC SQL WHENEVER SQLERROR GOTO EndOfProcessing;
EXEC SQL WHENEVER SQL_WARNING GOTO EndOfProcessing;
EXEC SQL WHENEVER NOT FOUND GOTO EndOfProcessing;

// print UCS2 string. Only characters in the range [0, 0xFF]
// are faithfully printed. Others are printed as '?'.
void print_UCS2_string(wchar_t* data, int len)
{
    for ( int i=0; i<len; i++ ) {
        if ( data[i] <= 0xFF )
            printf("%c", (char)data[i]);
        else
            printf("?");
    }
    printf("\n");
}

// print a single-byte string.
void print_singlebyte_string(char* data, int len)
{
    for ( int i=0; i<len; i++ )
        printf("%c", data[i]);
    printf("\n");
}

```

Example A-10. Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data (page 3 of 5)

```

// assume the sql text is in :hv_sql_stmt
void execute_one_statement()
{
    EXEC SQL SET NAMETYPE 'NSK';

    strcpy (in_sqlda,"          ");
    strcpy (out_sqlda,"          ");
    strcpy(in_sqlda,"selargs");
    in_sqlda[13] = '\0';
    strcpy(out_sqlda,"selcols");
    out_sqlda[13] = '\0';

    // Allocate the descriptor for input parameters
    hv_desc_max = 1;
    EXEC SQL ALLOCATE DESCRIPTOR :in_sqlda WITH MAX :hv_desc_max;

    // Allocate the descriptor for output values
    hv_desc_max = 6;
    EXEC SQL ALLOCATE DESCRIPTOR :out_sqlda WITH MAX
:hv_desc_max;

    // Prepare the statement
    EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

    // Describe the SQL descriptor area for input parameter
    EXEC SQL DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR
:in_sqlda;

    // Describe the SQL descriptor area for SELECT values
    EXEC SQL DESCRIBE OUTPUT sqlstmt USING SQL DESCRIPTOR :out_sqlda;

    // Get the input count
    j = 0;
    EXEC SQL GET DESCRIPTOR :in_sqlda :j = COUNT;

    if (j > 0)
    {
        // Get the type, character set name and length of the input
        EXEC SQL GET DESCRIPTOR :in_sqlda VALUE :j
                :data_type = TYPE,
                :data_len = LENGTH,
                :charset_name = CHARACTER_SET_NAME;

        // Set up the input value based on character set name
        if ( strcmp(charset_name, SQLCHARSETSTRING_KANJI) == 0 ) {
            wchar_t temp[21];
            //0123456789012345678901234567890123456789
            strcpy((char*)temp, "Japan");
            temp[20] = 0; // add the wide-char NULL for wcscopy
            wcscopy(hv_input_in_KANJI, (wchar_t*)temp);
            EXEC SQL SET DESCRIPTOR :in_sqlda VALUE :j
                    VARIABLE_DATA = :hv_input_in_KANJI;
        } else
    }

```

Example A-10. Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data (page 4 of 5)

```

        if ( strcmp(charset_name, SQLCHARSETSTRING_KSC5601) == 0 )
        {
            wchar_t temp[21];
                                //0123456789012345678901234567890123456789
            strcpy((char*)temp, "Korea");
            temp[20] = 0; // add the wide-char NULL for wcscopy
            wcscopy(hv_input_in_KSC5601, (wchar_t*)temp);
            EXEC SQL SET DESCRIPTOR :in_sqllda VALUE :j
                                VARIABLE_DATA = :hv_input_in_KSC5601;
        } else
            return;
    }

    // Execute the statement using the SQLDAs
    EXEC SQL EXECUTE sqlstmt USING SQL DESCRIPTOR :in_sqllda
        INTO SQL DESCRIPTOR :out_sqllda;

    // Get the count of the number of output values
    EXEC SQL GET DESCRIPTOR :out_sqllda :degree = COUNT;

    // Get the ith output value
    for (i=1; i<=degree; i++ ) {

        // Get the info about the output value
        EXEC SQL GET DESCRIPTOR :out_sqllda VALUE :i
                                :data_type = TYPE,

                                :charset_name = CHARACTER_SET_NAME,
                                :return_len = RETURNED_LENGTH,
                                :return_oct_len = RETURNED_OCTET_LENGTH;

        // Get and print out the output value based on the character set name
        if ( strcmp(charset_name, SQLCHARSETSTRING_KANJI) == 0 ) {
            EXEC SQL GET DESCRIPTOR :out_sqllda VALUE :i
                                :hv_output_in_KANJI = VARIABLE_DATA;
            print_singlebyte_string((char*)hv_output_in_KANJI,
return_oct_len);
        } else
            if ( strcmp(charset_name, SQLCHARSETSTRING_KSC5601) == 0 )
            {
                EXEC SQL GET DESCRIPTOR :out_sqllda VALUE :i
                                :hv_output_in_KSC5601 = VARIABLE_DATA;
                print_singlebyte_string((char*)hv_output_in_KSC5601, return_oct_len);
            } else return;
    }

    // Deallocate the prepared statement and the SQLDAs
    EXEC SQL DEALLOCATE prepare sqlstmt;
    EXEC SQL DEALLOCATE DESCRIPTOR :in_sqllda;
    EXEC SQL DEALLOCATE DESCRIPTOR :out_sqllda;

    return;

EndOfProcessing:
    handle_error();
}
void main()

```

Example A-10. Using SQL Descriptor Areas to Select SQL/MP KANJI and KSC5601 Data (page 5 of 5)

```
{
    /* input in KANJI and output in KSC5601 */
    strcpy(hv_sql_stmt, "select wordInKsc5601 FROM words WHERE \
wordInKanji = ? ;");
    execute_one_statement();

    /* input in KSC5601 and output in KANJI */
    strcpy(hv_sql_stmt, "select wordInKanji FROM words WHERE \
wordInKsc5601 = ? ;");
    execute_one_statement();
}
```

Using SQL Descriptors to Select UCS2 Data

[Example A-11](#) selects UCS2 data from an SQL/MX table by using SQL descriptor areas.

Example A-11. Using SQL Descriptors to Select UCS2 Data (page 1 of 5)

```

/* -----
   Description:  Illustrate setting and getting of character
                  set related description items, comparison of
                  UCS2 host variables with ISO88591 columns and
                  retrieval of ISO88591 data to UCS2 host
                  variables (relaxation).

   Statements:   ALLOCATE DESCRIPTOR
                  PREPARE
                  DESCRIBE
                  SET DESCRIPTOR
                  OPEN
                  FETCH
                  DEALLOCATE PREPARE
                  DEALLOCATE DESCRIPTOR
                  WHENEVER
                  GET DIAGNOSTICS
----- */
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <wchar.h>

EXEC SQL MODULE CSDYNCURSORM NAMES ARE ISO88591;
long  SQLCODE;
char  SQLSTATE_OK[6]  = "00000";
char  SQLSTATE_NODATA[6] = "02000";

EXEC SQL
BEGIN DECLARE SECTION;
    char  SQLSTATE[6];
    int   hv_desc_max;
    int   i, j;
    char  in_sqllda[13];
    char  out_sqllda[13];
    char  hv_sql_stmt[255];
    long  degree;
    long  data_type;
    long  data_len;
    long  return_len;
    VARCHAR charset_name[129];

    char  CHARACTER SET UCS2 hv_input_in_UCS2[21];
    VARCHAR CHARACTER SET UCS2 hv_output_in_UCS2[21];

EXEC SQL END DECLARE SECTION;

```

Example A-11. Using SQL Descriptors to Select UCS2 Data (page 2 of 5)

```

void handle_error()
{
    EXEC SQL BEGIN DECLARE SECTION;
        long i,num, hv_cond_num,hv_sqlcode;
        char hv_sqlstate[6], hv_table_name[129],hv_column_name[129];
        char hv_message_text[256];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR GOTO errexit;
    exec sql get diagnostics :num = NUMBER;

    for (i=1;i<=num;i++) {
        EXEC SQL get diagnostics exception :i
            :hv_cond_num = CONDITION_NUMBER,
            :hv_sqlstate = RETURNED_SQLSTATE,
            :hv_table_name = TABLE_NAME,
            :hv_column_name = COLUMN_NAME,
            :hv_message_text = MESSAGE_TEXT,
            :hv_sqlcode = SQLCODE;
        hv_sqlstate[5] = 0;
        printf("condition number: %d\n", hv_cond_num);
        printf("sqlstate: %s\n", hv_sqlstate);
        printf("table name: %s\n", hv_table_name);
        printf("column name: %s\n", hv_column_name);
        printf("message text: %s\n", hv_message_text);
        printf("sqlcode: %ld\n", hv_sqlcode);
        printf("\n");
    }

    return;

errexit:
    printf("\nError in the error handler.  SQLCODE = %d\n",
        SQLCODE);
    exit(1);
}

EXEC SQL WHENEVER SQLERROR GOTO EndOfProcessing;

// print UCS2 string. Only characters in the range [0, 0xFF]
// are faithfully printed. Others are printed as '?'.

void print_UCS2_string(wchar_t* data, int len)
{
    for ( int i=0; i<len; i++ ) {
        if ( data[i] <= 0xFF )
            printf("%c", (char)data[i]);
        else
            printf("?");
    }
}

```

Example A-11. Using SQL Descriptors to Select UCS2 Data (page 3 of 5)

```

// assume the sql text is in :hv_sql_stmt
void execute_one_statement()
{
    strcpy (in_sqlda,"          ");
    strcpy (out_sqlda,"          ");
    strcpy(in_sqlda,"selargs");
    in_sqlda[13] = '\0';
    strcpy(out_sqlda,"selcols");
    out_sqlda[13] = '\0';

    // Allocate the descriptor for input parameters
    hv_desc_max = 1;
    EXEC SQL ALLOCATE DESCRIPTOR :in_sqlda WITH MAX :hv_desc_max;

    // Allocate the descriptor for output values
    hv_desc_max = 6;
    EXEC SQL ALLOCATE DESCRIPTOR :out_sqlda WITH MAX \
        :hv_desc_max;

    // Prepare the cursor statement
    EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

    // Declare the dynamic cursor from the prepared statement
    EXEC SQL DECLARE cur_name CURSOR FOR sqlstmt;

    // Describe the SQL descriptor area for input parameter
    EXEC SQL DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR
        :in_sqlda;

    // Describe the SQL descriptor area for SELECT values
    EXEC SQL DESCRIBE OUTPUT sqlstmt USING SQL DESCRIPTOR
        :out_sqlda;

    // Get the input count
    j = 0;
    EXEC SQL GET DESCRIPTOR :in_sqlda :j = COUNT;

```

Example A-11. Using SQL Descriptors to Select UCS2 Data (page 4 of 5)

```

if (j > 0)
{
    // Get the type, character set name and length of the
    // input
    EXEC SQL GET DESCRIPTOR :in_sqlda VALUE :j
           :data_type = TYPE,
           :data_len = LENGTH,
           :charset_name = CHARACTER_SET_NAME;

    // Set up the input value based on character set name. Use
    // the relaxation feature to compare an UCS2 string with
    // an ISO88591 or UCS2 column.
    if ( strcmp(charset_name, SQLCHARSETSTRING_UNICODE) == 0 ||
        strcmp(charset_name, SQLCHARSETSTRING_ISO88591) == 0 )
    {
        //01234567890123
        wcscpy(hv_input_in_UCS2, L"Houston ");
        data_len = wcslen(hv_input_in_UCS2)+1;
        EXEC SQL SET DESCRIPTOR :in_sqlda VALUE :j
               VARIABLE_DATA = :hv_input_in_UCS2;
    } else
        return;
}

// Open the cursor using the input SQLDA
EXEC SQL OPEN cur_name USING SQL DESCRIPTOR :in_sqlda;

// Fetch the first row into output SQLDA
EXEC SQL FETCH cur_name INTO SQL DESCRIPTOR :out_sqlda;

// Get the count of the number of output values
EXEC SQL GET DESCRIPTOR :out_sqlda :degree = COUNT;

while ( strcmp(SQLSTATE, SQLSTATE_NODATA) != 0 ) {

```

Example A-11. Using SQL Descriptors to Select UCS2 Data (page 5 of 5)

```

// Get the ith output value
for (i=1; i<=degree; i++ ) {

    // Get the info about the output value. Assume it is CHARACTER data.
    EXEC SQL GET DESCRIPTOR :out_sqlda VALUE :i
        :data_type = TYPE,
        :charset_name = CHARACTER_SET_NAME;

    // Get and print out the output values. Use the relaxation feature
    // again to reuse the output host variable ":hv_output_in_UCS2" for
    // ISO88591 columns.

    if ( strcmp(charset_name, SQLCHARSETSTRING_UNICODE) == 0 ||
        strcmp(charset_name, SQLCHARSETSTRING_ISO88591) == 0 )
    {
        EXEC SQL GET DESCRIPTOR :out_sqlda VALUE :i
            :hv_output_in_UCS2 = VARIABLE_DATA,
            :return_len = RETURNED_LENGTH;

        print_UCS2_string(hv_output_in_UCS2, return_len);

        if ( i < degree )
            printf(", ");
    }
    printf("\n");

    EXEC SQL FETCH cur_name INTO SQL DESCRIPTOR :out_sqlda;
}

// Deallocate the prepared statement and the SQLDAs
EXEC SQL close cur_name;
EXEC SQL DEALLOCATE prepare sqlstmt;
EXEC SQL DEALLOCATE DESCRIPTOR :in_sqlda;
EXEC SQL DEALLOCATE DESCRIPTOR :out_sqlda;

return;

EndOfProcessing:
    handle_error();
}

void main()
{
    /*test ISO88591 data.*/
    strcpy(hv_sql_stmt, "select custname, street FROM usa_customer \
        WHERE city = ? ;");
    execute_one_statement();

    /*test UCS2 data.*/
    strcpy(hv_sql_stmt, "select custname, street FROM international_customer \
        WHERE city = ? ;");
    execute_one_statement();
}

```

B C++ Sample Program

This appendix presents the steps shown in [Section 14, Transaction Management](#), as a C++ program.

Ensuring Data Consistency

[Example B-1](#) executes the steps shown in [Figure 14-1](#) on page 14-1.

Example B-1. Using TMF to Ensure Data Consistency (page 1 of 2)

```
/* -----
   Description:      Using TMF to ensure data consistency
   Statements:      BEGIN WORK
                   INSERT
                   COMMIT WORK
                   ROLLBACK WORK
                   WHENEVER
----- */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
EXEC SQL MODULE EXF92CPM NAMES ARE ISO88591;

char SQLSTATE_OK[6]="00000";
EXEC SQL BEGIN DECLARE SECTION;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

void end_prog();
EXEC SQL DECLARE CATALOG 'samdbcat';
EXEC SQL DECLARE SCHEMA 'persnl';

EXEC SQL WHENEVER SQLERROR CALL end_prog;

class jobsql {
// Class member host variables
EXEC SQL BEGIN DECLARE SECTION;
    unsigned NUMERIC (4) memhv_jobcode;
    VARCHAR memhv_jobdesc[19];
EXEC SQL END DECLARE SECTION;

public:
    jobsql(){}; // Default constructor
    ~jobsql(){}; // Default destructor

// Member function to prompt for the job code
void getcode(){ cout << "Enter job code: " ;
                cin >> memhv_jobcode; }
```

Example B-1. Using TMF to Ensure Data Consistency (page 2 of 2)

```

// Member function to prompt for the job description
void getdesc(){ cout << "Enter job description: " ;
                cin >> memhv_jobdesc; }

// Member function to put the host variables into the table.
// The host variables are referenced in member functions
// defined within the same class.
void putjob(){
    EXEC SQL
        INSERT INTO job
        VALUES (:memhv_jobcode, :memhv_jobdesc);
}

// Member functions for begin work, commit work, rollback work
static void bw() { EXEC SQL BEGIN WORK; }
static void cw() { EXEC SQL COMMIT WORK; }
static void rw() { EXEC SQL ROLLBACK WORK; }

}; // End of class definition for jobsql

int main()
{
    SQLSTATE[5]='\0';
    SQLSTATE_OK[5]='\0';
    cout << "Example begins and ends a transaction." << endl;

    jobsql mysql; // Instantiate a member of class jobsql

    // Prompt for member class data for table
    mysql.getcode();
    mysql.getdesc();

    jobsql::bw(); // Begin the transaction

    // Insert the data into the table
    mysql.putjob();
    jobsql::cw(); // Commit the transaction
    cout << "The insert is committed." << endl;
    return 0;
} // End of main

void end_prog()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    if (strcmp(SQLSTATE, SQLSTATE_OK) != 0) {
        cout << "The insert is rolled back." << endl;
        cout << "SQLSTATE: " << SQLSTATE << endl;
        EXEC SQL ROLLBACK WORK; // Rollback the changes
    }
    exit(1);
} // End of end_prog

```

COBOL Sample Programs

This appendix presents the steps shown in figures in previous sections as complete COBOL programs.

Using a Static SQL Cursor

[Example C-1](#) executes the steps shown in [Figure 6-2](#) on page 6-3.

COBOL

Example C-1. Using a Static SQL Cursor (page 1 of 3)

```
-----
* Description: Using a Static SQL Cursor
* Statements: Static DECLARE CURSOR
*             BEGIN WORK
*             OPEN
*             FETCH
*             Positioned UPDATE
*             CLOSE
*             COMMIT WORK
*             WHENEVER
*             GET DIAGNOSTICS
* -----
IDENTIFICATION DIVISION.
PROGRAM-ID.    Program-exF62.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate          pic x(5).
01 sqlcode           pic s9(9) comp.
01 hv-partnum        pic 9(4) comp.
01 hv-partdesc       pic x(18).
01 hv-price          pic s9(6)v9(2) comp.
01 hv-qty-available  pic s9(7) comp.
01 hv-num            pic s9(9) comp.
01 hv-sqlstate       pic x(5).
01 hv-tabname        pic x(128).
01 hv-colname        pic x(128).
01 hv-msgtxt         pic x(128).
01 in-partnum        pic 9(4) comp.
01 i                 pic s9(9) comp.
    EXEC SQL END DECLARE SECTION END-EXEC.
01 sqlstate-ok       pic x(5) value "00000".
01 sqlstate-nodata   pic x(5) value "02000".
01 sqlstate-save     pic x(5).
01 sqlcode-save      pic s9(9) comp.
```

COBOL**Example C-1. Using a Static SQL Cursor (page 2 of 3)**

```

PROCEDURE DIVISION.
START-LABEL.
    DISPLAY "This example uses a static cursor.".
    EXEC SQL WHENEVER SQLERROR GOTO sqlerrors END-EXEC.
    EXEC SQL DECLARE CATALOG 'samdbcat' END-EXEC.
    EXEC SQL DECLARE SCHEMA 'sales' END-EXEC.
* Declare static cursor.
    EXEC SQL DECLARE get_by_partnum CURSOR FOR
        SELECT partnum, partdesc, price, qty_available
        FROM parts
        WHERE partnum >= :in-partnum
        FOR UPDATE OF partdesc, price, qty_available
    END-EXEC.

* Read in-partnum from terminal.
    DISPLAY "Enter lowest part number to be retrieved: ".
    ACCEPT in-partnum.

* Begin the transaction.
    EXEC SQL BEGIN WORK END-EXEC.

* Open the cursor.
    EXEC SQL OPEN get_by_partnum END-EXEC.

* Fetch the first row of the result from table.

    EXEC SQL FETCH get_by_partnum
        INTO :hv-partnum, :hv-partdesc,
            :hv-price, :hv-qty-available
    END-EXEC.

* Update qty_available if qty_available is less than 1000.
    PERFORM UNTIL sqlstate = sqlstate-nodata
        IF hv-qty-available < 1000
            EXEC SQL UPDATE parts
                SET qty_available = qty_available + 100
                WHERE CURRENT OF get_by_partnum
            END-EXEC.
            DISPLAY "Update of part number: " hv-partnum
        END-IF

    EXEC SQL FETCH get_by_partnum
        INTO :hv-partnum, :hv-partdesc,
            :hv-price, :hv-qty-available
    END-EXEC.
END-PERFORM.

```

COBOL**Example C-1. Using a Static SQL Cursor (page 3 of 3)**

```

*
* Close the cursor.
    EXEC SQL CLOSE get_by_partnum END-EXEC.
* Commit any changes.
    EXEC SQL COMMIT WORK END-EXEC.

    IF sqlstate = sqlstate-ok
        DISPLAY "The program completed successfully.".

    STOP RUN.
*****
sqlerrors SECTION.
*****

    move sqlstate to sqlstate-save.
    move sqlcode to sqlcode-save.
    display "sqlerrors: " sqlstate ", " sqlcode.
    EXEC SQL WHENEVER SQLError CONTINUE END-EXEC.

    IF sqlstate not = sqlstate-ok
        EXEC SQL GET DIAGNOSTICS
            :hv-num      = NUMBER
        END-EXEC.
        PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
            MOVE SPACES TO hv-msgtxt
            EXEC SQL GET DIAGNOSTICS EXCEPTION :i
                :hv-tabname  = TABLE_NAME,
                :hv-colname  = COLUMN_NAME,
                :hv-sqlstate = RETURNED_SQLSTATE,
                :hv-msgtxt   = MESSAGE_TEXT
            END-EXEC.
            DISPLAY "Table      : " hv-tabname
            DISPLAY "Column    : " hv-colname
            DISPLAY "SQLSTATE: " hv-sqlstate
            DISPLAY "Message   : " hv-msgtxt
        END-PERFORM
    END-IF.
    move sqlstate-save to sqlstate.
    move sqlcode-save to sqlcode.

*
    STOP RUN.

*****
END PROGRAM Program-exF62.
*****

```

Ensuring Data Consistency

[Example C-2](#) executes the steps shown in [Figure 14-2](#) on page 14-2

Example C-2. Using TMF to Ensure Data Consistency (page 1 of 2)

```

* -----
* Description: Using TMF to ensure data consistency
* Statements: SET TRANSACTION
*              BEGIN WORK
*              Searched UPDATE
*              COMMIT WORK
*              ROLLBACK WORK
*              WHENEVER
*              GET DIAGNOSTICS
* -----
IDENTIFICATION DIVISION.
PROGRAM-ID.    Program-exF92.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate          pic x(5).
    EXEC SQL END DECLARE SECTION END-EXEC.

01 sqlstate-ok       pic x(5) value "00000".

PROCEDURE DIVISION.
START-LABEL.
    DISPLAY "This example begins a transaction.".

    EXEC SQL DECLARE CATALOG 'samdbcat' END-EXEC.
    EXEC SQL DECLARE SCHEMA 'sales' END-EXEC.

* Set the attributes for the transaction.
    EXEC SQL
        SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
    END-EXEC.

* Start the transaction.
    EXEC SQL BEGIN WORK END-EXEC.

* Update the database by resetting customer credit.
    EXEC SQL
        UPDATE customer SET CREDIT = 'CR'
    END-EXEC.

```

Example C-2. Using TMF to Ensure Data Consistency (page 2 of 2)

```
IF sqlstate = sqlstate-ok
  DISPLAY "The transaction is committed."
  EXEC SQL COMMIT WORK END-EXEC.
ELSE
  DISPLAY "The transaction is rolled back."
  EXEC SQL ROLLBACK WORK END-EXEC.
END-IF.

STOP RUN.

*****
END PROGRAM Program-exF92.
*****
```

Using Argument Lists in Dynamic SQL

[Example C-3](#) executes the steps shown in [Figure 9-2](#) on page 9-4.

Example C-3. Using Argument Lists in Dynamic SQL (page 1 of 3)

```

*-----
*   Description:      Using Argument Lists
*   Statements:      PREPARE
*                   EXECUTE USING ARGUMENTS
*                   DEALLOCATE PREPARE
*                   WHENEVER
*                   GET DIAGNOSTICS
*-----
IDENTIFICATION DIVISION.
PROGRAM-ID.   Program-exF102.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate          pic x(5).
01 hv-empnum         pic 9(4) comp.
01 hv-first-name     pic x(15).
01 hv-last-name      pic x(20).
01 hv-salary         pic 9(6)v9(2) comp.
01 hv-salary-i       pic S9(4) comp.
01 hv-temp           pic 9(6)v9(2) display.

01 in-empnum         pic 9(4) comp.
01 hv-sql-stmt       pic x(256).

01 hv-num            pic S9(9) comp.
01 hv-sqlstate       pic x(5).
01 hv-msgtxt         pic x(128).
01 hv-tabname        pic x(128).
01 hv-colname        pic x(128).
01 i                 pic S9(9) comp.
    EXEC SQL END DECLARE SECTION END-EXEC.

01 sqlstate-ok       pic x(5) value "00000".
01 sqlstate-nodata   pic x(5) value "02000".

PROCEDURE DIVISION.
START-LABEL.
    DISPLAY "This example uses argument lists.".
    EXEC SQL WHENEVER SQLERROR GOTO sqlerrors END-EXEC.

```

Example C-3. Using Argument Lists in Dynamic SQL (page 2 of 3)

```

* Move statement with input variable to statement variable.
  MOVE "SELECT empnum, first_name, last_name, salary"
    & " FROM samdbcat.persnl.employee"
    & " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)"
  TO hv-sql-stmt.

* Prepare the statement.
  EXEC SQL PREPARE sqlstmt FROM :hv-sql-stmt END-EXEC.

* Initialize the input parameter in the WHERE clause.
  DISPLAY "Enter the employee number to be retrieved: ".
  ACCEPT in-empnum.

* Execute the prepared statement using the argument lists.
  EXEC SQL EXECUTE sqlstmt
    USING :in-empnum
    INTO :hv-empnum, :hv-first-name, :hv-last-name,
    :hv-salary INDICATOR :hv-salary-i END-EXEC.

* Process the output values.
  IF sqlstate = sqlstate-ok
    DISPLAY "Empnum is: " hv-empnum
    DISPLAY "First name is: " hv-first-name
    DISPLAY "Last name is: " hv-last-name
    IF hv-salary-i < 0
      DISPLAY "Salary is unknown"
    ELSE
      DIVIDE 100.0 INTO hv-salary GIVING hv-temp
      DISPLAY "Salary is: " hv-temp
  ELSE IF sqlstate = sqlstate-nodata
    DISPLAY "No row with employee number: " in-empnum.

* Deallocate the prepared statement.
  EXEC SQL DEALLOCATE PREPARE sqlstmt END-EXEC.

  STOP RUN.

```

Example C-3. Using Argument Lists in Dynamic SQL (page 3 of 3)

```

*****
sqlerrors SECTION.
*****

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

IF sqlstate not = sqlstate-ok
  EXEC SQL GET DIAGNOSTICS
    :hv-num      = NUMBER
  END-EXEC.
  PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
    MOVE SPACES TO hv-msgtxt
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv-tabname  = TABLE_NAME,
      :hv-colname  = COLUMN_NAME,
      :hv-sqlstate = RETURNED_SQLSTATE,
      :hv-msgtxt   = MESSAGE_TEXT
    END-EXEC.
    DISPLAY "Table      : " hv-tabname
    DISPLAY "Column    : " hv-colname
    DISPLAY "SQLSTATE: " hv-sqlstate
    DISPLAY "Message   : " hv-msgtxt
  END-PERFORM
END-IF.

STOP RUN.

*****
END PROGRAM Program-exF102.
*****

```

Using SQL Descriptor Areas in Dynamic SQL

[Example C-4](#) executes the steps shown in [Figure 10-2](#) on page 10-13.

Example C-4. Using Descriptor Areas With DESCRIBE (page 1 of 4)

```

* -----
* Description: Using SQL Descriptor Areas
* Statements: ALLOCATE DESCRIPTOR
*              PREPARE
*              DESCRIBE
*              SET DESCRIPTOR
*              EXECUTE
*              GET DESCRIPTOR
*              DEALLOCATE PREPARE
*              DEALLOCATE DESCRIPTOR
*              WHENEVER
*              GET DIAGNOSTICS
* -----
IDENTIFICATION DIVISION.
PROGRAM-ID.    Program-exF112.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 sqlstate          pic x(5).
01 hv-empnum         pic 9(4) comp.
01 hv-first-name     pic x(15).
01 hv-last-name      pic x(20).
01 hv-deptnum        pic 9(4) comp.
01 hv-jobcode        pic 9(4) comp.
01 hv-salary         pic 9(6)v9(2) comp.
01 hv-temp           pic 9(6)v9(2) display.
01 hv-sql-stmt.
    03 in-select      pic x(7) value "SELECT ".
    03 in-columns     pic x(80).
    03 in-from        pic x(80) value " FROM"
        & " samdbcat.persnl.employee"
        & " WHERE empnum = CAST(? AS NUMERIC(4) UNSIGNED)".
01 hv-prepare-stmt   pic x(170).
01 hv-num            pic s9(9) comp.
01 hv-sqlstate       pic x(5).
01 hv-msgtxt         pic x(128).
01 hv-tabname        pic x(128).
01 hv-colname        pic x(128).
01 sqlda-name        pic x(128).
01 in-empnum         pic 9(4) comp.
01 i                 pic s9(9) comp.
01 hv-desc-max       pic s9(9) comp.
01 hv-desc-value     pic s9(9) comp.
    EXEC SQL END DECLARE SECTION END-EXEC.
01 sqlstate-ok       pic x(5) value "00000".

```

Example C-4. Using Descriptor Areas With DESCRIBE (page 2 of 4)

```

PROCEDURE DIVISION.
START-LABEL.
    DISPLAY "This example uses SQL descriptor areas".
    EXEC SQL WHENEVER SQLERROR GOTO sqlerrors END-EXEC.

* Initialize the output variable in SELECT list.
    DISPLAY "Enter the columns to be retrieved,"
        & " separated by commas: ".
    ACCEPT in-columns.
    DISPLAY hv-sql-stmt.

* Allocate SQL descriptor area for input parameters.
    MOVE 1 TO hv-desc-max.
    EXEC SQL
        ALLOCATE DESCRIPTOR 'in_sqlda' WITH MAX :hv-desc-max
    END-EXEC.

* Allocate SQL descriptor area for output values.
    MOVE 6 TO hv-desc-max.
    EXEC SQL
        ALLOCATE DESCRIPTOR 'out_sqlda' WITH MAX :hv-desc-max
    END-EXEC.

* Prepare dynamic SQL statement.
    MOVE hv-sql-stmt TO hv-prepare-stmt.
    EXEC SQL
        PREPARE sqlstmt FROM :hv-prepare-stmt
    END-EXEC.

* Describe the SQL descriptor area for input parameters.
    EXEC SQL
        DESCRIBE INPUT sqlstmt USING SQL DESCRIPTOR 'in_sqlda'
    END-EXEC.

* Describe the SQL descriptor area for SELECT values.
    EXEC SQL
        DESCRIBE OUTPUT sqlstmt USING SQL DESCRIPTOR 'out_sqlda'
    END-EXEC.

* Initialize the input parameter in the WHERE clause
    DISPLAY "Enter the employee number to be retrieved: ".
    ACCEPT in-empnum.

* Set the value of the input parameter in
* the input SQL descriptor area
    MOVE 1 TO hv-desc-value.
    EXEC SQL
        SET DESCRIPTOR 'in_sqlda' VALUE :hv-desc-value
        VARIABLE_DATA = :in-empnum
    END-EXEC.

```

Example C-4. Using Descriptor Areas With DESCRIBE (page 3 of 4)

- * Execute the prepared statement using the SQL descriptor areas.

```
EXEC SQL
  EXECUTE sqlstmt
    USING SQL DESCRIPTOR 'in_sqlda'
    INTO SQL DESCRIPTOR 'out_sqlda'
END-EXEC.
```

- * Get the count of the number of output values.

```
EXEC SQL
  GET DESCRIPTOR 'out_sqlda' :hv-num = COUNT
END-EXEC.
```

- * Get the i-th output value in NAME field and save it.

```
PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
MOVE SPACES TO sqlda-name
EXEC SQL
  GET DESCRIPTOR 'out_sqlda' VALUE :i
  :sqlda-name = NAME
END-EXEC.
IF sqlda-name = "EMPNUM"
  EXEC SQL
    GET DESCRIPTOR 'out_sqlda' VALUE :i
    :hv-empnum = VARIABLE_DATA
  END-EXEC.
  DISPLAY "Empnum is:      " hv-empnum
ELSE
  IF sqlda-name = "FIRST_NAME"
    EXEC SQL
      GET DESCRIPTOR 'out_sqlda' VALUE :i
      :hv-first-name = VARIABLE_DATA
    END-EXEC.
    DISPLAY "First name is: " hv-first-name
  ELSE
    IF sqlda-name = "LAST_NAME"
      EXEC SQL
        GET DESCRIPTOR 'out_sqlda' VALUE :i
        :hv-last-name = VARIABLE_DATA
      END-EXEC.
      DISPLAY "Last name is:  " hv-last-name
    ELSE
      IF sqlda-name = "DEPTNUM"
        EXEC SQL
          GET DESCRIPTOR 'out_sqlda' VALUE :i
          :hv-deptnum = VARIABLE_DATA
        END-EXEC.
        DISPLAY "Department is: " hv-deptnum
      ELSE
        IF sqlda-name = "JOBCODE"
          EXEC SQL
            GET DESCRIPTOR 'out_sqlda' VALUE :i
            :hv-jobcode = VARIABLE_DATA
          END-EXEC.
          DISPLAY "Jobcode is:  " hv-jobcode
```

Example C-4. Using Descriptor Areas With DESCRIBE (page 4 of 4)

```

ELSE
  IF sqllda-name = "SALARY"
    EXEC SQL
      GET DESCRIPTOR 'out_sqllda' VALUE :i
      :hv-salary = VARIABLE_DATA
    END-EXEC.
    DIVIDE 100.0 INTO hv-salary GIVING hv-temp
    DISPLAY "Salary is: " hv-temp
  ELSE
    DISPLAY "Sqllda-name is      " sqllda-name
  END-IF
END-IF
END-IF
END-IF
END-IF
END-PERFORM.

EXEC SQL DEALLOCATE PREPARE sqlstmt END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR 'in_sqllda' END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR 'out_sqllda' END-EXEC.

IF sqlstate = sqlstate-ok
  DISPLAY "The program completed successfully."

STOP RUN.
*****
sqlerrors SECTION.
*****
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

IF sqlstate not = sqlstate-ok
  EXEC SQL GET DIAGNOSTICS
    :hv-num      = NUMBER
  END-EXEC.
  PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
    MOVE SPACES TO hv-msgtxt
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv-tabname = TABLE_NAME,
      :hv-colname = COLUMN_NAME,
      :hv-sqlstate = RETURNED_SQLSTATE,
      :hv-msgtxt   = MESSAGE_TEXT
    END-EXEC.
    DISPLAY "Table      : " hv-tabname
    DISPLAY "Column    : " hv-colname
    DISPLAY "SQLSTATE: " hv-sqlstate
    DISPLAY "Message   : " hv-msgtxt
  END-PERFORM
END-IF.
STOP RUN.
*****
END PROGRAM Program-exF112.
*****

```

Using a Dynamic SQL Cursor

[Example C-5](#) executes the steps shown in [Figure 10-2](#) on page 10-13.

Example C-5. Using a Dynamic SQL Cursor (page 1 of 3)

```

*-----
* Description: Using a Dynamic SQL Cursor
* Statements:  PREPARE
*              Dynamic DECLARE CURSOR
*              OPEN
*              FETCH
*              CLOSE
*              WHENEVER
*              GET DIAGNOSTICS
*-----

IDENTIFICATION DIVISION.
PROGRAM-ID.   Program-exF122.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 sqlstate           pic x(5).
01 hv-partnum         pic 9(4) comp.
01 hv-partdesc       pic x(18).
01 hv-price          pic s9(6)v9(2) comp.
01 hv-qty-available  pic s9(7) comp.
01 hv-num            pic s9(9) comp.
01 hv-sqlstate       pic x(5).
01 hv-msgtxt         pic x(128).
01 hv-tabname        pic x(128).
01 hv-colname        pic x(128).
01 curspec           pic x(255).
01 in-qty-available  pic s9(7) comp.
01 i                 pic s9(9) comp.

    EXEC SQL END DECLARE SECTION END-EXEC.

01 sqlstate-ok        pic x(5) value "00000".
01 sqlstate-nodata    pic x(5) value "02000".
01 print-line.
    03 print-order    pic x(19) value "Order part number: ".
    03 print-partnum  pic x(10).
    03 print-current  pic x(17) value "Current quality: ".
    03 print-qty      pic x(10).

```

Example C-5. Using a Dynamic SQL Cursor (page 2 of 3)

```

PROCEDURE DIVISION.
START-LABEL.
    DISPLAY "This example uses a dynamic cursor.".
    EXEC SQL WHENEVER SQLERROR GOTO sqlerrors END-EXEC.

    MOVE "SELECT partnum, partdesc, price, qty_available"
      & " FROM samdbcat.sales.parts"
      & " WHERE qty_available <= CAST(? AS NUMERIC(5))"
    TO curspec.

* Prepare cursor specification.
    EXEC SQL PREPARE cursor_spec FROM :curspec END-EXEC.

* Declare the dynamic cursor from the prepared statement.
    EXEC SQL
      DECLARE get_by_partnum CURSOR FOR cursor_spec
    END-EXEC.

* Initialize the parameter in the WHERE clause.
    DISPLAY "Enter the quantity to initiate the order: ".
    ACCEPT in-qty-available.

* Open the cursor using the values of the dynamic parameter.
    EXEC SQL
      OPEN get_by_partnum USING :in-qty-available
    END-EXEC.

* Fetch the first row of result from table.
    EXEC SQL
      FETCH get_by_partnum
      INTO :hv-partnum, :hv-partdesc,
          :hv-price,    :hv-qty-available
    END-EXEC.

* Fetch rest of the results from table.
    PERFORM UNTIL sqlstate = sqlstate-nodata
      MOVE hv-partnum TO print-partnum
      MOVE hv-qty-available TO print-qty
      DISPLAY print-line

      EXEC SQL FETCH get_by_partnum
        INTO :hv-partnum, :hv-partdesc,
            :hv-price,    :hv-qty-available
      END-EXEC.
    END-PERFORM.

* Close the cursor.
    EXEC SQL CLOSE get_by_partnum END-EXEC.

    IF sqlstate = sqlstate-ok
      DISPLAY "The program completed successfully.".
    STOP RUN.

```

Example C-5. Using a Dynamic SQL Cursor (page 3 of 3)

```

*****
sqlerrors SECTION.
*****

EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

IF sqlstate not = sqlstate-ok
  EXEC SQL GET DIAGNOSTICS
    :hv-num      = NUMBER
  END-EXEC.
  PERFORM VARYING i FROM 1 BY 1 UNTIL i > hv-num
    MOVE SPACES TO hv-msgtxt
    EXEC SQL GET DIAGNOSTICS EXCEPTION :i
      :hv-tabname  = TABLE_NAME,
      :hv-colname  = COLUMN_NAME,
      :hv-sqlstate = RETURNED_SQLSTATE,
      :hv-msgtxt   = MESSAGE_TEXT
    END-EXEC.
    DISPLAY "Table      : " hv-tabname
    DISPLAY "Column    : " hv-colname
    DISPLAY "SQLSTATE: " hv-sqlstate
    DISPLAY "Message   : " hv-msgtxt
  END-PERFORM
END-IF.

STOP RUN.

*****
END PROGRAM Program-exF122.
*****

```

Index

A

ALTER SQLMP ALIAS statement [8-3](#)
Argument lists, within dynamic SQL
 examples of [9-3](#)
 rowsets [12-5](#)
 sample program, COBOL [C-6](#)
 summary of statements [9-2](#)
Assignment statement [5-15](#)
Autocommit setting [14-4](#)
Automatic recompilation [8-12](#)

B

BEGIN WORK statement [6-11](#)
Buffers, with VARIABLE_POINTER [10-8](#)

C

C host variables [2-3](#)
 See also Host variables
 creating with INVOKE [3-31](#)
 data types [3-2](#)
 example of [3-28](#)
C preprocessing directives
 #define [15-11](#)
 #include [15-9](#)
 #line [15-12](#)
c89 utility
 examples of [15-42](#), [15-45](#)
 options for SQL/MX [15-37](#)
 using to compile C/C++ program in a
 single command [15-37](#)
 using to compile C/C++
 statements [15-41](#)
CALL statement [2-12](#)
CAST function
 converting character string to date-time
 value
 COBOL [4-16](#)

CAST function (continued)
 C/C++ [3-26](#)
 converting date-time column to
 character string
 COBOL [4-15](#)
 C/C++ [3-25](#)
Catalog declaration [2-7](#)
Character host variables
 C [3-2](#)
 COBOL [4-2](#), [4-10](#)
Character sets, examples
 C [3-40](#)
 COBOL [4-30](#)
CLOSE statement [11-6](#)
 description of [6-11](#)
 examples of [6-11](#)
COBOL compiler and linker, running [16-23](#)
COBOL host variables
 See also Host variables
 creating with INVOKE [4-22](#)
 data types [4-2](#)
COBOL preprocessor
 description of [16-13](#)
 functions [16-9](#)
 OSS-hosted [16-13](#)
 Windows-hosted [16-18](#)
COBOL program compilation [16-1](#)
Comments
 host language [2-2](#)
 naming an SQL statement
 in a COBOL program [16-10](#)
 in a C/C++ program [15-12](#)
 SQL [2-2](#)
COMMIT WORK statement [6-11](#)
Compilation
 COBOL [16-23](#)
 C++ [15-1](#)

- Compilation of modules
 - embedded module definitions [15-30](#), [16-25](#)
 - module definition files [15-34](#), [16-29](#)
- Compilation of programs
 - description of [1-13](#)
 - embedded module definitions [15-3](#), [16-3](#)
 - examples
 - build C application [15-50](#)
 - build COBOL application [16-43](#)
 - build C/C++ with SQL [15-47](#)
 - deploy static SQL to RDF system [15-54](#)
 - develop native C/C++ with SQL on OSS [15-49](#)
 - quick builds and mxcmp defaults in one-file deployment [15-52](#)
 - module definition files [15-6](#), [16-6](#)
- Compile-time name resolution [8-6](#)
- Compile-time specified length [12-4](#)
- Compound statements
 - assignment of [5-15](#)
 - description of [5-1](#), [5-13](#)
 - examples
 - grouping statements, C [5-14](#)
 - grouping statements, COBOL [5-14](#)
 - using SELECT INTO statement, C [5-15](#)
 - using SELECT INTO statement, COBOL [5-15](#)
 - general syntax [5-13](#)
 - IF statement [5-16](#)
 - limitation [5-13](#)
- CONTROL statements
 - See also Scope
 - behavior and use guidelines [2-14](#)
 - CONTROL QUERY DEFAULT [2-7](#)
 - CONTROL QUERY SHAPE [2-7](#)
 - CONTROL TABLE [2-7](#)
 - description of [2-12](#)
- CONTROL statements (continued)
 - dynamic [2-13](#)
 - static [2-13](#)
- CREATE SQLMP ALIAS statement [1-1](#), [8-2](#)
- CREATE TABLE, example of [3-35](#)
- Cursor position
 - after DELETE [6-9](#)
 - after FETCH [6-6](#)
 - after UPDATE [6-8](#)
 - operations affecting [6-15](#)
- Cursor specification, preparing [11-4](#)
- Cursors
 - changing position [6-15](#)
 - declaration statement [2-7](#)
 - description of [6-1](#)
 - dynamic
 - description of [11-1](#)
 - sample program, C [A-17](#)
 - sample program, COBOL [C-13](#)
 - sensitivity [6-16](#)
 - stability of [6-15](#)
 - static
 - description of [6-1](#)
 - sample program, C [A-1](#)
 - sample program, COBOL [C-1](#)
 - steps for using dynamic cursor in C [11-2](#)
 - steps for using dynamic cursor in COBOL [11-3](#)
- C++ class, SQL Declare Section [3-29](#)
- C++ host variables [2-3](#)
- C++ run-time library, default [15-29](#)
- C/C++ compiler and linker, running [15-28](#)
- C/C++ host variables, data types [3-13](#)
- C/C++ preprocessor
 - description of [15-8](#)
 - functions of [15-9](#)
 - OSS-hosted [15-16](#)

C/C++ preprocessor (continued)
 running with mxsqlc command [15-4](#),
 [15-7](#)
 Windows-hosted [15-21](#)
C/C++ program compilation [15-1](#)

D

Data consistency
 committing database changes [14-8](#)
 declaring host variables [14-3](#)
 default attributes for transactions [14-6](#)
 description of [1-11](#)
 error testing, examples of [14-7](#)
 examples of
 sample program, COBOL [C-4](#)
 sample program, C++ [B-1](#)
 grouping statements within
 transactions [14-7](#)
 rolling back database changes [14-8](#)
 starting transactions [14-6](#)
 steps for ensuring
 C [14-1](#)
 COBOL [14-2](#)
 system-initiated transaction [14-7](#)
Data Control Language (DCL) statements,
nonexecutable [2-7](#)
Data conversion
 between SQL and C data types [3-11](#)
 between SQL and COBOL data
 types [4-8](#)
Data Definition Language (DDL) statements
 embedding considerations [2-12](#)
 list of [2-8](#)
Data description clauses, COBOL [4-18](#)
Data Manipulation Language (DML)
statements
 description of [1-4](#)
 embedding considerations [2-12](#)
 list of [2-10](#)
 static SQL cursors [6-1](#)

Data Manipulation Language (DML)
statements (continued)
 using no cursor with [5-1](#)
 using rowsets [7-6](#), [7-29](#)
Data types
 conversion between C and SQL [3-11](#)
 conversion between COBOL and
 SQL [4-8](#)
 corresponding SQL and C [3-7](#)
 date-time
 COBOL [4-13](#)
 C/C++ [3-23](#)
 dynamic cursors [11-7](#)
 static cursors [6-12](#)
 decimal
 C [3-18](#)
 COBOL [4-12](#)
 fixed-length character
 C [3-14](#)
 COBOL [4-11](#)
 floating-point [3-21](#), [6-14](#)
 interval
 COBOL [4-13](#)
 C/C++ [3-23](#)
 dynamic cursors [11-7](#)
 static cursors [6-12](#)
 numeric
 C [3-18](#)
 COBOL [4-12](#)
 picture
 C [3-18](#)
 COBOL [4-12](#)
 syntax
 C host variables [3-2](#)
 COBOL host variables [4-2](#)
 variable-length character
 COBOL [4-12](#)
 C/C++ [3-16](#)
Database object names, referencing [1-1](#)

DATE host variable

description of [3-4](#), [4-3](#)format example [3-23](#), [4-13](#)

Date-time data

assigning to char array

COBOL [4-15](#)C/C++ [3-25](#)

casting

COBOL [4-15](#), [4-16](#)C/C++ [3-25](#), [3-26](#)dynamic cursors [11-7](#)

format

COBOL [4-13](#)C/C++ [3-23](#)

inserting from char array

COBOL [4-16](#)C/C++ [3-26](#)

inserting or updating

COBOL [4-15](#)C/C++ [3-25](#)nonstandard SQL/MP DATETIME data types [3-25](#), [4-15](#)

selecting

COBOL [4-14](#)C/C++ [3-24](#)static cursors [6-12](#)static rowsets [7-9](#)

Date-time examples

dynamic cursors

nonstandard SQL/MP DATETIME,
C [11-9](#)standard date-time, C [11-7](#)

inserting

nonstandard SQL/MP DATETIME
value [3-26](#), [4-16](#)standard date-time value [3-25](#),
[4-15](#)

selecting

nonstandard SQL/MP DATETIME
value [3-25](#), [4-15](#)

Date-time examples (continued)

standard date-time value [3-24](#),
[4-14](#)

static cursors

nonstandard SQL/MP DATETIME,
C [6-13](#)standard date-time, C [6-12](#)

static rowsets

C [7-9](#)COBOL [7-10](#)Date-time host variables [3-4](#), [4-13](#)

DDL

See Data Definition Language (DDL)
statementsDEALLOCATE PREPARE statement [11-7](#)

Debugging

Native Inspect [15-67](#)Visual Inspect [15-67](#)

Decimal data

assigning to char array [3-19](#)inserting from char array [3-19](#)

DECLARE CURSOR declaration

coding [6-4](#)description of [6-4](#)

example

read-only cursor [6-4](#)updatable cursor [6-5](#)DECLARE CURSOR statement,
syntax [11-4](#)Default transaction attributes [14-6](#)Default value setting for dynamic SQL [9-8](#)DEFINEs, using for table name [8-3](#)

DELETE statement

deleting a single-row, example [5-12](#)deleting multiple-rows, example [5-12](#)deleting rows with rowset arrays,
examples [7-20](#)positioned example [6-9](#)positioned form [6-9](#)privileges [5-12](#)

searched form

DELETE statement (continued)rowsets [7-19](#)syntax [5-12](#)**DESCRIBE statement**description of [12-10](#)INPUT form [10-4](#)OUTPUT form [10-7](#)**Descriptor**

See also Diagnostics area

allocating [10-3](#)deallocating [10-3](#)description of [10-1](#)input parameters [10-3](#)

output variables

description of [10-7](#)retrieving values of [10-7](#)setting input data values [10-4](#)setting input parameter [10-6](#)specifying output with dynamic cursor [11-10](#)

using descriptor areas

sample program, C [A-17](#)sample program, COBOL [C-9](#)**Diagnostics area [13-13](#)****Diagnostics statement [2-8](#)****DLL file**COBOL [16-18](#)C/C++ [15-21](#)**DML**

See Data Manipulation Language (DML)

Dynamic cursorsclosing [11-6](#)declaration of [11-4](#)declaring host variables for [11-4](#)description of [11-1](#)initializing input parameters [11-5](#)opening [11-5](#)preparing query expression [11-4](#)processing retrieved values [11-6](#)**Dynamic cursors (continued)**retrieving values [11-5](#)

sample program

C [A-15](#)COBOL [C-13](#)steps for using in C [11-2](#)steps for using in COBOL [11-3](#)using [1-8](#)using descriptor areas, example [11-10](#)**Dynamic input parameters [11-5](#)****Dynamic recompilation, SQL/MX [15-30](#), [16-25](#)****Dynamic rowsets**DESCRIBE statement [12-10](#)description of [12-1](#)descriptor fields, setting [12-5](#)example of [A-26](#)GET DESCRIPTOR statement [12-9](#)matching compile-time specified length
with execution-time length [12-4](#)preparing an SQL statement [12-2](#)rowset parameter, specifying [12-3](#)use restrictions [12-1](#)using SET DESCRIPTOR
statement [12-5](#)with argument lists [12-5](#)**Dynamic SQL**advantages of [9-1](#)allocating SQL descriptor area for input
parameters [10-14](#)allocating SQL descriptor area for
output variables [10-15](#)assigning input value to DATA within
SQL descriptor area [10-17](#)constructing statement from
input [10-14](#)cursors [11-1](#)deallocating resources for prepared
statement [10-20](#)deallocating SQL descriptor area [9-7](#),
[10-20](#)

Dynamic SQL (continued)

- declaring host variable for statement [10-14](#)
- default value setting [9-8](#)
- describing input parameters [10-16](#)
- describing output parameters [10-16](#)
- description of [1-7](#), [9-1](#)
- executing the statement [10-18](#)
- getting COUNT of item descriptor areas [10-19](#)
- getting VALUE of output within SQL descriptor area [10-19](#)
- input parameters in [10-3](#)
- parameters [10-3](#)
- preparing the statement [9-5](#), [10-15](#)
- rowsets [12-1](#)
- setting INDICATOR for null within SQL descriptor area [10-17](#)
- setting values of input parameters by position [10-17](#)
- statements for dynamic cursors, summary [11-1](#)
- statements with descriptors, summary [10-1](#)
- steps for using SQL descriptor area [10-12](#), [10-13](#)
- using descriptor areas
 - description of [1-7](#)
 - sample program, C [A-17](#)
 - sample program, COBOL [C-9](#)
- with arguments
 - deallocating resources for prepared statement [9-7](#)
 - declaring host variable for statement [9-4](#)
 - executing the prepared statement [9-6](#)
 - floating-point variables [9-2](#)
 - input parameters [9-2](#)
 - moving statement into host variable [9-5](#)

Dynamic SQL (continued)

- output variables [9-2](#)
- sample program, COBOL [C-6](#)
- steps for using [9-3](#)
- summary [9-2](#)
- using in a C program, example of [9-3](#)
- using in a COBOL program, example of [9-4](#)

E

ecobol utility

- examples of [16-38](#), [16-41](#)
- options for SQL/MX [16-33](#)
- using to compile COBOL program in a single command [16-32](#)

Embedded module definitions

- compiling, COBOL [16-25](#)
- compiling, C/C++ [15-30](#)

Embedded SQL

- advantages of [1-1](#)
- coding in program [2-1](#)
- description of [1-2](#)
- executable SQL statements [2-4](#)
- general syntax of [2-1](#)
- host variable declarations [2-2](#)
- MODULE directive [2-2](#)
- nonexecutable SQL statements [2-4](#)
- placement of [2-5](#)

Embedded SQL application, running

- COBOL [16-48](#)
- C/C++ [15-64](#)

Environment variable

- MXCMP [15-34](#), [16-29](#)
- MXCMPUM [15-34](#), [16-29](#)

Error

- SQLCODE values [13-6](#)

Error conditions [13-1](#)

ETK

See HP Enterprise Toolkit (ETK)

Exception conditions

- checking SQLCODE [13-5](#)
- checking SQLSTATE [1-10](#), [13-1](#)
- description of [1-10](#)
- using GET DIAGNOSTICS [13-13](#)
- using WHENEVER [13-8](#)

Exception declaration [2-7](#)Exception handling [13-1](#)

EXECUTE IMMEDIATE statement, setting dynamic SQL default values with [9-8](#)

F

FETCH statement

- description of [6-6](#)
- examples of [6-6](#), [6-10](#), [7-12](#)
- loop processing [6-10](#)
- selecting rowsets [7-12](#)
- syntax for transferring values [11-5](#)

Fixed-length character data

- declaring char array [3-14](#)
- inserting and updating [3-15](#)
- inserting from char array [3-14](#)
- selecting into char array [3-14](#)

Fixed-point data types, assigning [3-20](#)

Floating-point data types

- assigning [3-21](#)
- changing the format [3-22](#)
- conversion between Tandem and IEEE formats [3-22](#)
- IEEE [3-21](#)
- restrictions [3-22](#)
- Tandem [3-21](#)
- using [6-14](#)

Floating-point formats [3-6](#)Floating-point variables [9-2](#)**G**

GET DESCRIPTOR statement [10-8](#), [12-9](#)

GET DIAGNOSTICS statement

- example of [1-11](#), [13-14](#)
- using [1-11](#)

Globally placed modules

- coexistence with locally placed modules [17-4](#)
- generating [17-3](#)
- syntax [17-4](#)
- system-wide setting [17-5](#)

Grouping

- C example, INVENTORY modules [17-21](#)
- COBOL example, INVENTORY modules [17-22](#)
- description of [17-21](#)
- setting up example, C/C++ or COBOL [17-21](#)

Guardian environment

- building C program to use SQL/MX [15-60](#)
- building C++ program to use SQL/MX [15-62](#)
- running SQL/MX programs in
 - C [15-58](#)
 - COBOL [16-44](#)
- using TACL macro to execute commands, C++ [15-63](#)

H

Hardware service interruptions, handling [13-1](#)

Host language

- compiler options [1-14](#)
- preprocessor [1-13](#)

Host variables

- char [3-2](#)
- conversion between SQL and C data types [3-11](#)

Host variables (continued)

data types

C [3-2](#)COBOL [4-2](#)

date-time

COBOL [4-3](#), [4-13](#)C/C++ [3-4](#), [3-23](#)

declaring

COBOL [4-1](#)C/C++ [3-1](#)dynamic cursors [11-4](#)examples [1-3](#)static rowsets [7-2](#)

defined

COBOL [4-1](#)C/C++ [3-1](#)extended data types [3-9](#)floating-point format [3-6](#)in C [2-3](#)in C++ [2-3](#)

indicator variables

C [3-12](#)COBOL [4-10](#)

initializing host variables specified in

DECLARE CURSOR [6-5](#)

input, defined

COBOL [4-1](#)C/C++ [3-1](#)

interval

COBOL [4-3](#), [4-13](#)C/C++ [3-4](#), [3-23](#)

naming conventions

C [3-12](#)COBOL [4-9](#)NCHAR [3-3](#)NCHAR VARYING [3-4](#)

numeric

C [3-5](#)COBOL [4-6](#)

Host variables (continued)

output, defined

COBOL [4-1](#)C/C++ [3-1](#)PROTOTYPE host variables [5-17](#), [8-4](#)referring to within C structure [3-28](#)rowset arrays [7-4](#)

syntax

specifying in C [3-12](#)specifying in COBOL [4-9](#)

using data members of a C++

class [3-29](#)using for table names [8-4](#)VARCHAR [3-3](#)

variable-length character

declaration [3-16](#)HP Enterprise Toolkit (ETK) [15-29](#)

I

IEEE floating-point format [3-6](#)IF statement [5-16](#)

Indicator variables

description of [3-29](#), [4-19](#)

host variable specification

C [3-12](#)COBOL [4-9](#)syntax [7-4](#)inserting NULL [3-13](#), [3-30](#), [4-10](#), [4-19](#),
[7-4](#), [8-4](#)

return value

COBOL [4-10](#)C/C++ [3-13](#)syntax [7-4](#)testing for NULL [3-30](#), [4-20](#)testing for truncated value [3-30](#), [4-20](#)updating columns to null [5-11](#)INDICATOR_POINTER, exclusive use
of [12-9](#)

INSERT statement

- inserting date-time

- nonstandard SQL/MP DATETIME

- value [5-8](#)

- standard date-time value [5-8](#)

- inserting from host variables [5-4](#), [7-14](#), [7-34](#)

- inserting from rowset arrays [7-14](#), [7-17](#)

- inserting from rowset-derived table [7-34](#)

- inserting interval values

- C [5-8](#)

- COBOL [5-9](#)

- inserting NULL

- C [5-6](#)

- COBOL [5-7](#)

- rowset arrays [7-15](#)

- rowset-derived table [7-36](#)

- inserting rows, C [5-5](#)

- inserting timestamp value [7-17](#)

Inspect, for debugging [15-67](#)

Interval data

- dynamic cursors [11-7](#), [11-8](#)

- inserting

- COBOL [4-17](#)

- C/C++ [3-27](#)

- selecting

- COBOL [4-17](#)

- C/C++ [3-27](#)

- static cursors [6-13](#)

- static rowsets [7-9](#)

- updating

- C [3-28](#)

- COBOL [4-18](#)

INTERVAL host variable

- description of [3-5](#)

- format example [3-26](#)

INVOKE directive

- C example [3-39](#)

- creating host variables

INVOKE directive (continued)

- C [3-31](#)

- COBOL [4-22](#)

- example of C structure generated by [3-36](#)

- generating C structures [3-34](#)

- generating COBOL structures [4-23](#)

- generating indicator variables

- C [3-37](#)

- COBOL [4-27](#)

- NULL STRUCTURE clause

- C example [3-37](#)

- COBOL example [4-28](#)

- PREFIX clause

- C example [3-37](#)

- COBOL example [4-27](#)

- SUFFIX clause

- C example [3-37](#)

- COBOL example [4-27](#)

ISO88591 character set, selecting into UCS2 host variable [3-42](#)

Isolation level setting [14-5](#)

L

Large buffers and VARIABLE_POINTER [10-8](#)

Late name resolution

- purpose of [8-6](#)

- syntax [5-17](#)

- using host variables [8-4](#)

- using host variables and DEFINES [8-3](#)

Locally placed modules

- coexistence with globally placed modules [17-4](#)

- co-locating with application [17-5](#)

- generating [17-3](#)

- system-wide setting [17-4](#)

LOCK TABLE statement [2-11](#)

M

MAP DEFINE [8-3](#)

Mapping from logical to physical object names [1-2](#)

MDFs

See Module definition files (MDFs)

Module compilation

definition files [15-34](#), [16-29](#)

embedded module definitions [15-30](#), [16-25](#)

Module creation, SQL/MX methods

COBOL [16-2](#)

comparison of [1-12](#)

C/C++ [15-2](#)

Module definition files (MDFs)

compiling COBOL [16-29](#)

compiling C/C++ [15-34](#)

generating [17-7](#)

MODULE directive placement [2-2](#)

Module file errors [15-66](#), [16-49](#)

Module management

See also Grouping, Targeting, and Versioning

description of [17-1](#)

file naming [15-67](#), [16-50](#)

grouping [17-21](#)

influencing behavior [17-7](#)

naming

description of [17-8](#)

effect of name change, example of [17-12](#)

name components [17-9](#)

name length, example of [17-10](#)

targeting [17-12](#)

tasks [17-6](#)

versioning [17-19](#)

MPLOC attribute declaration [2-7](#)

MXCI

See SQL/MX conversational interface (MXCI)

mxcmp command

COBOL [16-29](#)

C/C++ [15-34](#)

examples of [15-36](#), [16-31](#)

syntax [15-35](#), [16-30](#)

MXCMP environment variable [15-34](#), [16-29](#)

MXCMPUM environment variable [15-34](#), [16-29](#)

mxCompileUserModule

embedded definitions [15-30](#)

examples of [15-34](#), [16-29](#)

syntax [15-31](#), [16-26](#)

mssqlc command, using to run SQL/MX C/C++ preprocessor, example of [15-28](#)

mssqlco command, using to run SQL/MX COBOL preprocessor, examples of [16-17](#), [16-23](#)

N

Name qualification [8-5](#)

Name resolution

compile-time [8-6](#)

description of [8-1](#)

distributed database considerations [8-7](#)

late name resolution [8-6](#)

OLT optimization, DEFINE names and PROTOTYPE host variables [8-5](#)

RDF considerations [8-7](#)

NAMETYPE attribute declaration [2-7](#)

Naming conventions for host variables

C [3-12](#)

COBOL [4-9](#)

National character set

NCHAR host variable [3-3](#)

NCHAR VARYING [3-4](#)

Native Inspect, for debugging [15-67](#)

NCHAR host variable [3-3](#)

NCHAR VARYING host variable [3-4](#)

Network interruptions, handling [13-1](#)

nmcobol utility

examples [16-38](#), [16-41](#)

nmcobol utility (continued)

- options for SQL/MX [16-33](#)
- using to compile COBOL program in a single command [16-32](#)

NULL

- inserting [3-30](#)
- inserting multiple rows with
 - using indicator host variable array [7-15](#)
 - using indicator host variable array in rowset-derived table [7-36](#)
- keyword instead of indicator variable [4-20](#)
- retrieving using NULL predicate [3-31](#), [4-21](#)
- testing for [3-30](#), [4-20](#)

NULL STRUCTURE clause [3-37](#)Numeric host variables [3-5](#), [4-12](#)**O**OBEY command files, for creating database objects [2-12](#)Object name qualification [8-5](#)Object names, referencing [1-1](#)Object naming statements [2-10](#), [2-11](#)OLT optimization considerations [8-5](#)OLTP programs, recommended recompilation settings [8-13](#)

OPEN statement

- description of [6-5](#)
- syntax [11-5](#)

Open System Services (OSS)

- COBOL preprocessing in [16-13](#)
- developing native C/C++ with SQL on [15-49](#)

P

Parameters

- dynamic SQL with arguments
 - description of [9-2](#)
 - input [9-2](#)

Parameters (continued)

- output [9-2](#)
- SELECT columns [9-2](#)
- using DESCRIBE INPUT [9-2](#)
- using DESCRIBE OUTPUT [9-2](#)
- dynamic SQL with descriptor areas
 - description of [10-3](#)
 - input [10-3](#)
 - output [10-7](#)
 - SELECT columns [10-7](#)
 - using DESCRIBE INPUT [10-3](#)
 - using DESCRIBE OUTPUT [10-7](#)

Performance

- declaring host variables [3-12](#)
 - COBOL [4-1](#)
 - C/C++ [3-1](#)
- improving with rowsets [1-8](#)
- INVOKE statement
 - C [3-31](#)
 - COBOL [4-22](#)
- OLT optimization considerations [8-5](#)
- OLTP settings [8-13](#)
- recompiling [15-30](#), [16-25](#)
- rowsets [1-8](#), [7-1](#)
- similarity check [8-8](#)
- using a cursor [6-2](#)

PICTURE clause

- fixed-length character data [4-11](#)
- numeric data [4-12](#)
- variable-length character data [4-12](#)

Positioned DELETE statement [6-9](#)Positioned UPDATE statement [6-8](#)

PREPARE statement

- compiling dynamic embedded SQL [12-2](#)
- syntax [11-4](#)

PREPARE string, specifying rowset parameter [12-3](#)

Preprocess in Guardian environment

C [15-60](#)C++ [15-62](#)

Preprocessor

interpretation of environment variables,
example of [17-7](#)

OSS-hosted

COBOL [16-9](#)C/C++ [15-8](#), [15-17](#)

Windows-hosted

COBOL [16-18](#)C/C++ [15-21](#)

Program compilation

C and C++ [15-2](#)COBOL [16-3](#), [16-6](#)C/C++ [15-6](#)embedded module definitions [15-3](#),
[16-3](#)module definition files [15-6](#), [16-6](#)

Program management

description of [17-1](#)managing files [17-3](#)Program management, debugging [15-67](#)

PROTOTYPE host variables

example of [5-18](#)syntax [8-4](#)using as table names [5-18](#)**Q**

Query execution plans

displaying [15-67](#), [16-51](#)displaying for all statements [15-69](#)displaying for one statement [15-68](#)wild cards [15-69](#)**R**

RDF

See Remote Database Facility (RDF)

Record descriptions, COBOL [4-9](#)Redefinition timestamp, tables [8-9](#)

Remote Database Facility (RDF)

deploying static SQL to [15-54](#)object naming considerations [8-7](#)ROLLBACK WORK statement [6-11](#)

Rowset arrays

as input for SELECT statements,
examples of [7-11](#)description of [7-2](#)examples of [7-4](#)inserting rows from [7-14](#)selecting rows into [7-6](#)specifying size and row ID [7-20](#)syntax [7-4](#)updating rows with [7-17](#)using for input [7-5](#)using for output [7-6](#)using in DML statements [7-6](#)ROWSET FOR clause [7-20](#)

Rowset parameter

basic dynamic rowset, example [12-3](#)mixing scalar and rowset host variables,
example [12-3](#)specifying [12-3](#)using FOR INPUT SIZE and KEY BY,
example [12-4](#)

Rowsets

See also Dynamic rowsets, Static
rowsets, Rowset arraysdifference between static and
dynamic [12-1](#)

dynamic SQL

description of [12-1](#)

descriptor fields

syntax [12-5](#)example of [A-26](#)selecting into rowset arrays [7-6](#)

specifying rowset arrays

example of [7-4](#)syntax [7-4](#)

Rowset-derived tables

- available from static rowsets only [12-1](#)
- deleting rows by using [7-39](#)
- description of [7-28](#)
- inserting rows from [7-34](#)
- limiting the size of [7-35](#)
- updating rows by using [7-37](#)

ROWSET_IND_LAYOUT_SIZE [12-8](#)ROWSET_SIZE [12-6](#)

ROWSET_VAR_LAYOUT_SIZE

- description of [12-6](#)
- minimum values table [12-7](#)

Run-time errors, avoiding [15-65](#)Run-time library, C++ default [15-29](#)**S**Schema declaration [2-7](#)

Scope

- flow control [2-13](#)
- line order [2-13](#)

Searched DELETE statement [5-12](#)Searched UPDATE statement [5-9](#)

SELECT statement

- examples of [5-2](#), [7-11](#)
- multiple-row using rowsets [7-6](#)
- primary key in WHERE clause [5-2](#)
- single-row [5-2](#)
- using rowset arrays as input [7-10](#)
- using to specify a cursor [6-1](#)

SET DESCRIPTOR statement

- examples of [10-4](#), [10-6](#)
- using with dynamic rowsets [12-5](#)

SET MPLOC statement [2-11](#)SET NAMETYPE statement [2-10](#)SET TRANSACTION statement [14-3](#)

Similarity check

- automatic recompilation [8-12](#)
- criteria [8-10](#)
- enabling, disabling [8-9](#)
- purpose of [8-8](#)

Simple statements [5-1](#)SQL cursors, dynamic [11-1](#)

SQL Declare Section

- COBOL [4-1](#)
- C/C++ [3-1](#)
- placement of [2-2](#)

SQL descriptor area

- description of [10-2](#)
- getting information from [10-7](#)
- setting information in [10-4](#)
- steps for using [10-12](#), [10-13](#)

SQL item descriptor [10-2](#)

SQL statements

- coding guidelines [2-1](#)
- dynamic SQL [2-8](#)
- executable [2-4](#)
- executable in C++ [2-5](#)
- nonexecutable restrictions [2-4](#)
- preparing with dynamic rowsets [12-2](#)

SQLCODE

- declaring [13-5](#)
- examples of checking [13-6](#)
- using ERROR command [13-8](#)
- values [13-5](#)
- within SQL/MX messages [13-7](#)

SQLCODE Values [13-6](#)SQLMX_PREPROCESSOR_VERSION,
using to generate module definition
files [17-7](#)

SQLSTATE

- checking [1-10](#), [13-4](#)
- declaring [13-1](#)
- values [13-2](#)
- within SQL/MX messages [13-8](#)

SQL/MX compiler

- environment variable for backward
compatibility [15-36](#), [16-31](#)
- invoking [15-41](#), [16-37](#)
- mxcmp command to compile module
definition file [15-35](#), [16-30](#)

SQL/MX compiler (continued)

running

COBOL [16-25](#)C/C++ [15-30](#)specifying alternate location [15-34](#),
[16-29](#)SQL/MX conversational interface
(MXCI) [1-1](#), [13-7](#)

SQL/MX preprocessor

C [15-8](#)COBOL [16-9](#)functions [15-9](#)invoking [15-40](#), [16-37](#)OSS-hosted, COBOL [16-13](#)OSS-hosted, C/C++ [15-16](#)Windows-hosted, COBOL [16-18](#)Windows-hosted, C/C++ [15-21](#)

SQL/MX programs

building and running in Guardian
environment [15-58](#), [15-60](#), [15-62](#),
[16-44](#), [16-45](#)

compilation methods

COBOL [16-2](#)C/C++ [15-2](#)description of [1-12](#)running [15-65](#), [16-49](#)

Static cursors

authority to use [6-14](#)closing [6-11](#)declaration of [6-4](#)declaring host variables for query
expression [6-4](#)description of [1-5](#), [6-1](#)

examples

date-time, nonstandard [5-4](#), [6-13](#)date-time, standard [5-3](#), [6-12](#), [7-9](#)interval [5-4](#), [6-13](#)fetching within transaction [6-14](#)initializing host variables [6-5](#)lock mode considerations [6-15](#)

Static cursors (continued)

opening [6-5](#)processing retrieved values [6-7](#)retrieving values [6-6](#)

sample program

C [A-1](#)COBOL [C-1](#)statements using [6-1](#)steps for using [6-2](#), [6-3](#)

Static rowsets

See also Rowset arrays

C example of [7-3](#)COBOL example of [7-3](#)considerations for size [7-3](#)declaring host variable arrays [7-2](#)description of [7-1](#), [7-2](#)

limiting size of input rowset

examples of [7-22](#), [7-23](#)when declaring a cursor [7-23](#)limiting size of output rowset, examples
of [7-24](#)restricting size [7-20](#)

rowset-derived tables

specifying, syntax for [7-28](#)using in DML statements [7-29](#)selecting with rowset cursor [7-12](#)C [7-12](#)COBOL [7-13](#)specifying row ID [7-20](#)static cursor, example of [7-9](#)syntax diagram for host variable
array [7-3](#)using the index identifier, examples
of [7-25](#)Stored procedures (SPJ) statement [2-12](#)

Syntax

C host variable data types [3-2](#)COBOL host variable data types [4-2](#)SYSTEM_DEFAULTS table [8-6](#)

T

Table names

- PROTOTYPE host variables [8-4](#)
- SQL/MP, DEFINE names [8-3](#)
- SQL/MP, Guardian names [8-2](#)
- SQL/MP, SQL/MP aliases [8-2](#)
- SQL/MX, logical names [8-2](#)

TACL macro

- for C Guardian application [15-61](#)
- for COBOL Guardian application [16-46](#)
- for C++ Guardian application [15-63](#)

Tandem floating-point format [3-6](#)

Targeting

C example

- using build subdirectory [17-15](#)
- using Module TableSet (MTSS) [17-13](#)

COBOL example

- using build subdirectory [17-18](#)
- using Module TableSet (MTSS) [17-16](#)

description of [17-12](#)

effect of target attribute [17-13](#)

TIME host variable

- description of [3-4](#)
- length of C target arrays for [3-24](#)

TIMESTAMP host variable

- description of [3-5](#)
- length of C target arrays for [3-24](#)

Timestamp value, inserting [7-17](#)

Transaction

- attributes, example of [14-3](#)
- control statements [2-10](#), [14-1](#)
- isolation levels [14-6](#)

U

UCS2 character set

- fetching into VARCHAR host variable [3-41](#), [4-31](#)

UCS2 character set (continued)

- host variable, selecting ISO88591 character set data [3-42](#)
- selecting into VARCHAR host variable [3-41](#), [4-31](#)

UNLOCK TABLE statement [2-11](#)

UPDATE statement

- examples of [6-8](#), [7-18](#)
- positioned form [6-8](#)
- privileges [5-10](#)
- searched form
 - rowsets [7-17](#)
 - syntax [5-9](#)
- updating a single-row
 - C searched example [5-10](#)
 - COBOL searched example [5-10](#)
- updating multiple-rows
 - C searched example [5-11](#)
 - COBOL searched example [5-11](#)
- updating to NULL with indicator variable [5-11](#)
- using host variables [6-8](#)

UPDATE STATISTICS statement

- description of [2-12](#)
- embedding considerations [2-12](#)

Utilities statement [2-12](#)

V

VARCHAR host variable [3-3](#), [3-16](#)

Variable-length character data

- declaring [3-16](#)
- inserting or updating [3-17](#)

VARIABLE_POINTER

- exclusive use of [12-9](#)
- large buffers [10-8](#)

Versioning

- C setup example [17-20](#)
- COBOL setup example [17-20](#)
- description of [17-19](#)
- guidelines [17-20](#)

View names

using DEFINES [8-3](#)

using host variables [8-4](#)

Visual Inspect, for debugging [15-67](#)

W

Warning

SQLCODE values [13-6](#)

WHENEVER statement

avoiding infinite loops [13-10](#)

examples of [1-10](#), [13-9](#)

NOT FOUND condition [13-8](#)

precedence of conditions [13-9](#)

scope of [13-9](#)

SQLERROR condition [13-8](#)

SQL_WARNING condition [13-9](#)

using [1-10](#)

Windows

COBOL DLL file [16-18](#)

COBOL preprocessor, syntax [16-19](#)

C/C++ DLL file [15-21](#)

C/C++ preprocessor, syntax [15-23](#)

Z

ZCLIDLL [16-5](#), [16-8](#)

ZCLISRL [16-5](#), [16-8](#)

Special Characters

-Wmxcmp, command to invoke SQL/MX compiler [15-41](#), [16-37](#)

-Wsqlmx, command to invoke SQL/MX preprocessor [15-40](#), [16-37](#)