



Embedded SQL™/C Programmers Guide

Open Client™

15.7

DOCUMENT ID: DC37695-01-1570-01

LAST REVISED: April 2012

Copyright © 2012 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book	ix	
CHAPTER 1	Introduction	1
	Embedded SQL overview	1
	Embedded SQL features.....	2
	Transact-SQL support in Embedded SQL.....	2
	Getting started.....	3
	Using the examples.....	4
	Backward compatibility	4
	Creating and running an Embedded SQL program.....	4
	How the precompiler processes your applications	5
	Multiple Embedded SQL source files	6
	Precompiler compatibility	6
	Precompiler-generated files	6
CHAPTER 2	General Information	7
	Five tasks of an Embedded SQL program	7
	Simplified Embedded SQL program.....	8
	General rules for Embedded SQL.....	9
	Statement placement	9
	Comments	10
	Identifiers.....	10
	Quotation marks	10
	Reserved words	10
	Variable naming conventions	10
	Scoping rules.....	11
	Statement batches	11
	Embedded SQL constructs	11
CHAPTER 3	Communicating with Adaptive Server Enterprise.....	13
	Scoping rules: SQLCA, SQLCODE, and SQLSTATE	14
	Declaring SQLCA	14
	Multiple SQLCAs	14

	SQLCA variables	15
	Accessing SQLCA variables	15
	SQLCODE within SQLCA	16
	Declaring SQLCODE as a standalone area.....	16
	Using SQLSTATE	17
	Obtaining SQLSTATE codes and error messages.....	18
	Summary.....	18
CHAPTER 4	Using Variables.....	19
	Declaring variables.....	19
	Using datatypes.....	20
	Using type definitions	22
	Using #define	24
	Declaring an array.....	24
	Declaring unions and structures.....	26
	Using host variables.....	27
	Host input variables.....	28
	Host result variables.....	28
	Host status variables.....	29
	Host output variables.....	29
	Using indicator variables.....	29
	Indicator variables and server restrictions.....	30
	Using host variables with indicator variables.....	30
	Host variable conventions	32
	Using arrays.....	33
	Multiple arrays.....	33
	Scoping rules	34
	Datatypes and Adaptive Server Enterprise	34
	Converting datatypes	36
CHAPTER 5	Connecting to Adaptive Server Enterprise	39
	Connecting to a server.....	39
	user	40
	password.....	40
	connection_name	40
	server	40
	connect example	41
	Changing the current connection	41
	Establishing multiple connections	41
	Naming a connection.....	42
	Using Adaptive Server Enterprise connections	43
	Disconnecting from a server	44

CHAPTER 6	Using Transact-SQL Statements	47
	Transact-SQL statements in Embedded SQL.....	47
	exec sql syntax.....	47
	Invalid statements	48
	Transact-SQL statements that differ in Embedded SQL.....	48
	Selecting rows.....	48
	Selecting one row.....	49
	Selecting multiple rows using arrays.....	49
	Selecting multiple rows using cursors	53
	Using stored procedures	66
	Grouping statements.....	69
	Grouping statements by batches.....	69
	Grouping statements by transactions.....	70
CHAPTER 7	Using Dynamic SQL.....	73
	Dynamic SQL overview.....	74
	Dynamic SQL protocol	75
	Method 1: Using execute immediate.....	76
	Method 1 examples.....	76
	Method 2: Using prepare and execute	77
	prepare.....	78
	execute.....	79
	Method 2 example.....	79
	Method 3: Using prepare and fetch with a cursor	80
	prepare.....	81
	declare.....	81
	open.....	81
	fetch and close	82
	Method 3 example.....	83
	Method 4: Using prepare and fetch with dynamic descriptors	84
	Method 4 dynamic descriptors	84
	Dynamic descriptor statements	85
	Method 4 example using SQL descriptors	86
	About SQLDAs.....	89
	Method 4 example using SQLDAs	90
	Summary.....	92
CHAPTER 8	Handling Errors.....	93
	Testing for errors.....	94
	Using SQLCODE.....	94
	Testing for warning conditions	94
	Trapping errors with whenever.....	95
	whenever testing conditions	96

whenever actions	97
Using get diagnostics	97
Writing routines to handle warnings and errors.....	98
Precompiler-detected errors.....	99

CHAPTER 9 Improving Performance with Persistent Binding 101

About persistent binding.....	102
When binding occurs.....	103
Programs that can benefit from persistent binding.....	104
Scope of persistent bindings	105
Precompiler options for persistent binding	105
The -p option	105
The -b option	105
Which option to use: -p, -b, or both	106
Scope of the -p and -b precompiler options	106
Overview of rules for persistent binding	106
Statements that can use persistent binding	107
Persistent binding in statements without a cursor	107
Persistent binding in statements with a cursor	108
Guidelines for using persistent binding	113
Notes on the binding of host variables	114
Subscripted arrays	114
Scope of host variables	116

CHAPTER 10 Embedded SQL Statements: Reference Pages 119

allocate descriptor	121
begin declare section	122
begin transaction.....	123
close.....	124
commit.....	126
connect.....	127
deallocate cursor	129
deallocate descriptor	131
deallocate prepare	131
declare cursor (dynamic).....	132
declare cursor (static).....	133
declare cursor (stored procedure).....	135
declare scrollable cursor	136
delete (positioned cursor).....	138
delete (searched)	139
describe input (SQL descriptor)	141
describe input (SQLDA)	142
describe output (SQL descriptor)	143

describe output (SQLDA)	144
disconnect	145
exec.....	147
exec sql	150
execute.....	152
execute immediate	154
exit.....	155
fetch	155
fetch scrollable cursor	158
get descriptor	159
get diagnostics	161
include "filename".....	162
include sqlca	164
include sqlda	165
initialize_application	165
open (dynamic cursor)	167
open (static cursor)	168
prepare.....	170
rollback.....	172
select.....	172
set connection	174
set descriptor.....	175
thread exit	177
update	177
whenever.....	179

CHAPTER 11	Open Client and Open Server Configuration File	185
	Purpose of the Open Client and Open Server configuration file ..	185
	Accessing the configuration functionality	185
	Default settings	186
	Syntax for the Open Client and Open Server configuration file ...	187
	Sample programs.....	189
	Embedded SQL/C sample makefile on Windows.....	189
	Embedded SQL/C sample programs	190
	Embedded SQL program version for use with the -x option..	190
	Same Embedded SQL program with the -e option.....	192
	Summary.....	195

APPENDIX A	Precompiler Warning and Error Messages	197
------------	---	------------

APPENDIX B	Sample Code for Handling Large Text and Image Data	211
	Where to find other samples	211

text_image.sql	211
text_image.cp	212
Glossary	215
Index	223

About This Book

This book explains how to use Embedded SQL™ and the Embedded SQL precompiler with C applications. Sybase® Embedded SQL is a superset of Transact-SQL® that lets you place Transact-SQL statements in application programs written in languages such as C and COBOL.

The information in this guide is platform-independent. For platform-specific instructions on using Embedded SQL, see the *Open Client and Open Server Programmers Supplement* for your platform.

Audience

This guide is intended for application developers and others interested in Embedded SQL concepts and uses. To use this guide, you should:

- Be familiar with the information presented in the *Adaptive Server Enterprise Reference Manual*
- Have C programming experience

How to use this book

This book contains these chapters:

- Chapter 1, “Introduction,” presents a brief overview of Embedded SQL and describes its advantages and capabilities.
- Chapter 2, “General Information,” describes the parts of an Embedded SQL program and provides general rules for programming with Embedded SQL.
- Chapter 3, “Communicating with Adaptive Server Enterprise,” describes how to establish and use a communication area with SQLCA, SQLCODE, and SQLSTATE. This chapter also describes the system variables used in the communication area.
- Chapter 4, “Using Variables,” explains how to declare and use host and indicator variables in Embedded SQL. This chapter also describes arrays and explains datatype conversions.
- Chapter 5, “Connecting to Adaptive Server Enterprise,” explains how to use Embedded SQL to connect an application program to Adaptive Server® Enterprise and data servers, in general.

-
- Chapter 6, “Using Transact-SQL Statements,” describes how to use Transact-SQL in an Embedded SQL application program. This chapter describes how to select rows using arrays and batches, and how to group Transact-SQL statements.
 - Chapter 7, “Using Dynamic SQL,” describes how to create Embedded SQL statements that your application’s users can enter interactively at runtime.
 - Chapter 8, “Handling Errors,” describes return codes and the Embedded SQL precompiler’s facilities for detecting and handling errors.
 - Chapter 9, “Improving Performance with Persistent Binding,” describes how performance might benefit from using persistent binding and how to implement it.
 - Chapter 10, “Embedded SQL Statements: Reference Pages,” provides reference pages for each Embedded SQL statement.
 - Chapter 11, “Open Client and Open Server Configuration File,” explains how to use the external configuration file with Embedded SQL.
 - Appendix A, “Precompiler Warning and Error Messages,” lists precompiler and runtime messages.
 - Appendix B, “Sample Code for Handling Large Text and Image Data,” contains sample programs for Embedded SQL that demonstrates the use of host variables in handling large text and image data.

Related documents

You can see these books for more information:

- The *Open Server and SDK New Features for Windows, Linux, and UNIX*, which describes new features available for Open Server and the Software Developer’s Kit. This document is revised to include new features as they become available.
- The *Open Server Release Bulletin* for your platform contains important last-minute information about Open Server.
- The *Software Developer’s Kit Release Bulletin* for your platform contains important last-minute information about Open Client™ and SDK.
- The *jConnect™ for JDBC™ Release Bulletin* contains important last-minute information about jConnect.
- The *Open Client and Open Server Configuration Guide* for your platform contains information about configuring your system to run Open Client and Open Server.

- The *Open Client Client-Library/C Programmers Guide* contains information on how to design and implement Client-Library applications.
- The *Open Client Client-Library/C Reference Manual* contains reference information for Open Client Client-Library™.
- The *Open Server Server-Library/C Reference Manual* contains reference information for Open Server Server-Library.
- The *Open Client and Open Server Common Libraries Reference Manual* contains reference information for CS-Library, which is a collection of utility routines that are useful in both Client-Library and Server-Library applications.
- The *Open Server DB-Library/C Reference Manual* contains reference information for the C version of Open Client DB-Library™.
- The *Open Client and Open Server Programmers Supplement* for your platform contains platform-specific information for programmers using Open Client and Open Server. This document includes information about:
 - Compiling and linking an application
 - The sample programs that are included with Open Client and Open Server
 - Routines that have platform-specific behaviors
- The *Installation and Release Bulletin Sybase SDK DB-Library Kerberos Authentication Option* contains information about installing and enabling the MIT Kerberos security mechanism to be used on DB-Library. DB-Library only supports network authentication and mutual authentication in the Kerberos security mechanism.
- The *Open Client and Open Server International Developers Guide* provides information about creating internationalized and localized applications.
- The *Open Client Embedded SQL™/COBOL Programmers Guide* explains how to use Embedded SQL and the Embedded SQL precompiler with COBOL applications.
- The *jConnect for JDBC Programmers Reference* describes the jConnect for JDBC product and explains how to access data stored in relational database management systems.

-
- The *Adaptive Server® Enterprise ADO.NET Data Provider Users Guide* provides information on how to access data in Adaptive Server using any language supported by .NET, such as C#, Visual Basic .NET, C++ with managed extension, and J#.
 - The *Adaptive Server Enterprise ODBC Driver by Sybase Users Guide* for Microsoft Windows and UNIX, provides information on how to access data from Adaptive Server on Microsoft Windows and UNIX platforms, using the Open Database Connectivity (ODBC) Driver.
 - The *Adaptive Server Enterprise OLE DB Provider by Sybase Users Guide for Microsoft Windows* provides information on how to access data from Adaptive Server on Microsoft Windows platforms, using the Adaptive Server OLE DB Provider.
 - The *Adaptive Server Enterprise Database Driver for Perl Programmers Guide* provides information for Perl developers to connect to an Adaptive Server database and query or change information using a Perl script.
 - The *Adaptive Server Enterprise extension module for PHP Programmers Guide* provides information for PHP developers to execute queries against an Adaptive Server database.
 - The *Adaptive Server Enterprise extension module for Python Programmers Guide* provides information about Sybase-specific Python interface that can be used to execute queries against an Adaptive Server database.

Other sources of information

Use the Sybase® Product Documentation Web site to learn more about your product:

- The Sybase Product Documentation Web site is accessible using a standard Web browser. In addition to product documentation, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

To access the Sybase Product Documentation Web site, go to Product Documentation at <http://www.sybase.com/support/manuals/>.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ Finding the latest information on product certifications

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click Partner Certification Report.

- 3 In the Partner Certification Report filter select a product, platform, and timeframe and then click Go.
- 4 Click a Partner Certification Report title to display the report.

❖ **Finding the latest information on component certifications**

- 1 Point your Web browser to Availability and Certification Reports at <http://certification.sybase.com/>.
- 2 Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
- 3 Select Search to display the availability and certification report for the selection.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

- 1 Point your Web browser to Technical Documents at <http://www.sybase.com/support/techdocs/>.
- 2 Click MySybase and create a MySybase profile.

**Sybase EBFs and
software
maintenance**

❖ **Finding the latest information on EBFs and software maintenance**

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
- 3 Select a product.
- 4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.

-
- Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

Table 1: Syntax conventions

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in sans serif font.
<i>variable</i>	Variables, or words that stand for values that you fill in, are in <i>italics</i> .
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include the braces in the command.
[]	Brackets mean choosing one or more of the enclosed items is optional. Do not include the braces in the command.
()	Parentheses are to be typed as part of the command.
	The vertical bar means you can select only one of the options shown.
,	The comma means you can choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Open Client and Open Server documentation has been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at <http://www.sybase.com/accessibility>. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the documentation or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.



Introduction

This chapter includes the following topics to introduce Embedded SQL and the Embedded SQL precompiler.

Topic	Page
Embedded SQL overview	1
Embedded SQL features	2
Transact-SQL support in Embedded SQL	2
Getting started	3
Creating and running an Embedded SQL program	4
How the precompiler processes your applications	5

Embedded SQL overview

Embedded SQL is a superset of Transact-SQL that lets you place Transact-SQL statements in application programs written in languages such as C and COBOL.

Open Client™ Embedded SQL enables you to create programs that access and update Adaptive Server Enterprise data. Embedded SQL programmers write SQL statements directly into an application program written in a conventional programming language such as C or COBOL. A preprocessing program—the Embedded SQL precompiler—processes the completed application program, resulting in a program that the host language compiler can compile. The program is linked with Open Client Client-Library before it is executed.

Embedded SQL is one of the two programming methods Sybase provides for accessing Adaptive Server Enterprise. The other programming method is the call-level interface. With the call-level interface, you place Client-Library calls directly into an application program, then link with Client-Library.

You can place Embedded SQL statements anywhere in a host program and mix them with host language statements. All Embedded SQL statements must begin with the keywords `exec sql` and end with a semicolon (`;`).

You can use *host variables* in Embedded SQL statements to store data retrieved from Adaptive Server Enterprise and as parameters in Embedded SQL statements, such as in the `where` clause of a `select` statement. In dynamic SQL, host variables can also contain text for Embedded SQL statements.

After you write an Embedded SQL program, run it through the precompiler, which translates the Embedded SQL statements into Client-Library function calls.

Embedded SQL features

Embedded SQL provides several advantages over a call-level interface:

- Embedded SQL is easy to use because it is simply Transact-SQL with some added features that facilitate using it in an application.
- It is an ANSI/ISO-standard programming language.
- It requires less coding to achieve the same results as a call-level approach.
- Embedded SQL is essentially identical across different host languages. Programming conventions and syntax change very little. Therefore, to write applications in different languages, you need not learn new syntax.
- The precompiler can optimize execution time by generating stored procedures for the Embedded SQL statements.

Transact-SQL support in Embedded SQL

Transact-SQL is the set of SQL commands described in the *Adaptive Server Enterprise Reference Manual*. With the exception of `print`, `readtext`, and `writetext`, all Transact-SQL statements, functions, and control-of-flow language are valid in Embedded SQL. You can develop an interactive prototype of your Embedded SQL application in Transact-SQL to facilitate debugging your application, then easily incorporate it into your application.

Most Adaptive Server Enterprise datatypes have an equivalent in Embedded SQL. Also, you can use host language datatypes in Embedded SQL. Many datatype conversions occur automatically when a host language datatype does not exactly match an Adaptive Server Enterprise datatype.

You can place host language variables in Embedded SQL statements wherever literal quotes are valid in Transact-SQL. Enclose the literal with either single (') or double (") quotation marks. For information on delimiting literals that contain quotation marks, see the *Adaptive Server Enterprise Reference Manual*.

Embedded SQL has several features that Transact-SQL does not have:

- *Automatic datatype conversion* occurs between host language types and Adaptive Server Enterprise types.
- *Dynamic SQL* lets you define SQL statements at runtime.
- *SQLCA*, *SQLCODE*, and *SQLSTATE* let you communicate between Adaptive Server Enterprise and the application program. The three entities contain error, warning, and informational message codes that Adaptive Server Enterprise generates.
- *Return code testing routines* detect error conditions during execution.

Getting started

Before attempting to run the precompiler, make sure that Client-Library version 12.5 or later is installed, since the precompiler uses it as the runtime library. Also, make sure Adaptive Server Enterprise version 12.5 or later is installed. If products are missing, contact your **System Administrator**.

Invoke the precompiler by issuing the appropriate command at the operating system prompt. See the *Open Client and Open Server Programmers Supplement* for your platform for details.

The precompiler command can include several flags that let you determine options for the precompiler, including the input file, login user name and password, invoking HA failover, and precompiler modes. The *Open Client and Open Server Programmers Supplement* contains operating system-specific information on precompiling, compiling, and linking your Embedded SQL application.

Using the examples

The examples in this guide use the pubs2 database. To run the examples, specify the pubs2 database with the Transact-SQL use statement.

Embedded SQL is shipped with several online examples. For information on running these examples, see the *Open Client and Open Server Programmers Supplement* for your platform.

Backward compatibility

The System 11 and later precompiler is compatible with precompilers that are SQL-89-compliant. However, you may have applications created with earlier Embedded SQL releases that are not ANSI-compliant. This precompiler uses most of the same Embedded SQL statements used in previous precompiler versions, but it processes them differently.

To migrate applications created for earlier precompiler releases:

- 1 Remove the following SQL statements and keywords from the application, as System 11 does not support them:
 - release *connection_name*
 - recompile
 - noparse
 - noproc
 - pcoptions
 - cancel

The release statement causes a precompiler error; the precompiler ignores the other keywords. The cancel statement causes a runtime error.

- 2 Use the System 11 and later precompiler to precompile the application again.

Creating and running an Embedded SQL program

Follow these steps to create and run an Embedded SQL application program:

- 1 Write the application program and include the Embedded SQL statements and variable declarations.
- 2 Save the application in a file with a *.cp* extension.
- 3 Precompile the application. If there are no severe errors, the precompiler generates a file containing your application program. The file has the same name as the original source file, with a different extension, depending on the requirements of your C compiler. For details, see the *Open Client and Open Server Programmers Supplement* for your platform.
- 4 Compile the new source code as you would compile a standard C program.
- 5 Link the compiled code with Client-Library.
- 6 If you specified the precompiler option to generate stored procedures, load them into Adaptive Server Enterprise by executing the generated script with *isql*.
- 7 Run the application program as you would any standard C program.

How the precompiler processes your applications

The Embedded SQL precompiler translates Embedded SQL statements into C data declarations and call statements. After precompiling, you can compile the resulting source program as you would any conventional C program.

The precompiler processes an application in two passes. In the first pass, the precompiler *parses* the Embedded SQL statements and variable declarations, checking the syntax and displaying messages for any errors it detects. If the precompiler detects no severe errors, it proceeds with the second pass, wherein it:

- Adds declarations for the precompiler variables, which begin with “_sql”. To prevent confusion, do not begin your variables’ names with “_sql”.
- Converts the text of the original Embedded SQL statements to comments.
- Generates stored procedures and calls to stored procedures if you set this option in the precompile command.
- Converts Embedded SQL statements to Client-Library calls. Embedded SQL uses Client-Library as a runtime library.

- Generates up to three files: a **target file**, an optional **listing file**, and an optional **isql script file**.

Note For detailed descriptions of precompiler command line options, see the *Open Client and Open Server Programmers Supplement* for your platform.

Multiple Embedded SQL source files

If the Embedded SQL application consists of more than one source file, the following statements apply:

- Connection names are unique and global to the entire application.
- Cursor names are unique for a given connection.
- Prepared statement names are global to the connection.
- Dynamic descriptors are global to the application.

Precompiler compatibility

Embedded SQL version 12.5 and later is completely ANSI SQL-89-compliant. Therefore, it is compatible with other precompilers that conform to ANSI-89 standards.

Precompiler-generated files

The target file is similar to the original input file, except that all SQL statements are converted to Client-Library runtime calls.

The listing file contains the input file's source statements, plus any informational, warning, or error messages.

The isql script file contains the precompiler-generated stored procedures. The stored procedures are written in Transact-SQL.

This chapter provides general information about Embedded SQL.

Topic	Page
Five tasks of an Embedded SQL program	7
General rules for Embedded SQL	9
Embedded SQL constructs	11

Five tasks of an Embedded SQL program

In addition to containing the host language code, an Embedded SQL program performs five tasks. Each Embedded SQL program must perform all these tasks to precompile, compile, and execute. Subsequent chapters discuss these five tasks.

- 1 Establish SQL communication using SQLCA, SQLCODE, or SQLSTATE.

Set up the SQL communication area (SQLCA, SQLCODE, or SQLSTATE) to provide a communication path between the application program and Adaptive Server Enterprise. These structures contain error, warning, and information message codes that Adaptive Server Enterprise and Client-Library generate. See Chapter 3, “Communicating with Adaptive Server Enterprise.”

- 2 Declare variables.
 - Identify host variables used in Embedded SQL statements to the precompiler. See Chapter 4, “Using Variables.”

- 3 Connect to Adaptive Server Enterprise.

Connect the application to Adaptive Server Enterprise. See Chapter 5, “Connecting to Adaptive Server Enterprise.”

- 4 Send Transact-SQL statements to Adaptive Server Enterprise.

- Send Transact-SQL statements to Adaptive Server Enterprise to define and manipulate data. See Chapter 6, “Using Transact-SQL Statements.”
- 5 Handle errors and return codes.
- Handle and report errors returned by Client-Library and Adaptive Server Enterprise using SQLCA, SQLCODE, or SQLSTATE. See Chapter 8, “Handling Errors.”

Simplified Embedded SQL program

Following is a simple Embedded SQL program. At this point, you need not understand everything shown in the program. Its purpose is to demonstrate the parts of an Embedded SQL program. The details are explained in subsequent chapters.

```
/* Establishing a communication area - Chapter 3 */

exec sql include sqlca;

main()
{

/* Declaring variables - Chapter 4 */

exec sql begin declare section;
CS_CHAR user[31], passwd[31];
exec sql end declare section;

/*Initializing error-handling routines - Chapter 8 */

exec sql whenever sqlerror call err_p();

/*Establishing Adaptive Server Enterprise connections
- Chapter 5 */

printf("\nplease enter your userid ");
gets(user);
printf("\npassword ");
gets(passwd);
exec sql connect :user identified by :passwd;

/* Issuing Transact-SQL statements - Chapter 6 */

exec sql update titles set price = price * 1.10;
```



```
exec sql commit work;

/* Closing server connections - Chapter 5 */

exec sql disconnect all;
}

/* Error-handling routines - Chapter 8 */

err_p()
{
    /* Print the error code and error message */

    printf("\nError occurred: code %d.\n%s",
        sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

General rules for Embedded SQL

The following rules apply to Embedded SQL statements in C programs:

- Embedded SQL statements begin with these keywords:

```
exec sql
```

- Embedded SQL statements must end with a semicolon:

```
exec sql sql_statement;
```

- Place `exec sql` at the beginning of the source line except when a C label precedes it:

```
[label:] exec sql sql_statement;
```

- Embedded SQL keywords are not case sensitive. `exec sql`, `EXEC SQL`, `Exec Sql`, or any other of case mix is equally valid. This manual shows Embedded SQL keywords in lowercase. For example:

```
exec sql commit work;
```

Statement placement

An application program can have Embedded SQL statements wherever C statements are valid.

Comments

Comments placed within Embedded SQL and C statements must follow one of two conventions.

The Transact-SQL convention is:

```
/* comments */
```

The ANSI convention is:

```
-- comments
```

Comments placed outside SQL statements must conform to C-programming conventions.

Identifiers

Identifiers are used as function or variable names within your application.

Quotation marks

Enclose literal character strings in Embedded SQL statements within single or double quotation marks. If a character string begins with a double quotation mark, end it with a double quotation mark. If a character string begins with a single quotation mark, end it with a single quotation mark.

Reserved words

Do not use C, Transact-SQL, or Embedded SQL reserved words except as intended by the languages.

You can write Embedded SQL keywords in uppercase, lowercase, or mixed case. This guide shows Embedded SQL keywords in lowercase.

Variable naming conventions

Embedded SQL variables must conform to C naming conventions. Do not place variable names within quotation marks. Applicable quotation marks are inserted automatically when the variable names are replaced with actual values.

While parsing the application, declarations are added for precompiler variables. These variables begin with “_sql”. So, to avoid confusion, do not begin variable names with “_sql”.

Scoping rules

Embedded SQL and precompiler-generated statements adhere to **host language** scoping rules. The whenever statement and cursor names are exceptions.

Statement batches

As in Transact-SQL, you can batch several SQL statements in a single `exec sql` statement. Batches are useful and more efficient when an application must execute a fixed set of Transact-SQL statements each time it runs.

For example, some applications create temporary tables and indexes when they start up. You could send these statements in a single batch. See the *Adaptive Server Enterprise Reference Manual* for rules about statement batches.

The following restrictions apply to statement batches:

- Statements in a batch cannot return results to the program. That is, a batch can contain no `select` statements.
- All statements in a batch must be valid Transact-SQL statements. You cannot place Embedded SQL statements such as `declare cursor` and `prepare` in a statement batch.
- The same rules that apply to Transact-SQL batches apply to Embedded SQL batches. For example, you cannot put a `use database` statement in an Embedded SQL batch.

Embedded SQL constructs

Table 2-1 displays valid constructs in Embedded SQL statements:

Table 2-1: Embedded SQL constructs

begin declare section	dump database
begin tran	dump tran

begin work	end declare section
checkpoint	exec <i>procedure_name</i>
close <i>cursor_name</i>	execute <i>name</i>
commit tran	execute immediate
commit work	fetch <i>cursor_name</i>
connect	grant
create database	include sqlca or include <i>filename</i>
create default	insert
create table	open <i>cursor_name</i>
create index	prepare <i>statement_name</i>
create unique index	revoke
create clustered index	rollback tran
create nonclustered index	rollback work
create unique clustered index	select
create unique nonclustered index	set
create proc	truncate
create rule	update
create trigger	use
create view	whenever <i>condition action</i>
declare cursor	
delete	
disconnect	
drop table default index proc rule trigger view	

Communicating with Adaptive Server Enterprise

This chapter explains how to enable an application program to receive status information from Adaptive Server Enterprise.

Topic	Page
Scoping rules: SQLCA, SQLCODE, and SQLSTATE	14
Declaring SQLCA	14
Declaring SQLCODE as a standalone area	16
Using SQLSTATE	17

To create a communication path and declare system variables to be used in communications from Adaptive Server Enterprise to the application, you must create one of three entities:

- A SQL Communication Area (SQLCA), which includes SQLCODE
- A standalone SQLCODE long integer
- A SQLSTATE character array

SQLCODE, SQLCA, and SQLSTATE are variables to be used in communication from Adaptive Server Enterprise to the application.

After Adaptive Server Enterprise executes each Embedded SQL statement, it stores return codes in SQLCA, SQLCODE, or SQLSTATE. An application program can access the variables to determine whether the executed SQL statement succeeded or failed.

Note The precompiler automatically sets SQLCA, SQLCODE, and SQLSTATE variables, which are critical for runtime access to the database. You need not initialize or modify them.

For details on detecting and handling errors, multiple error messages, and other return codes, see Chapter 8, “Handling Errors.”

Scoping rules: SQLCA, SQLCODE, and SQLSTATE

You can declare SQLCA anywhere in the application program where a C variable can be declared. The scope of the structure follows C scoping rules.

If you declare SQLCA, SQLCODE, or SQLSTATE within your file, each variable must be in scope for all executable Embedded SQL statements in the file. The precompiler generates code to set each of these status variables for each Embedded SQL statement. So, if the variables are not in scope, the generated code will not compile.

If you do not declare SQLCA, SQLCODE, or SQLSTATE within the file being passed to the precompiler, you must declare SQLCODE within a referenced file. The precompiler assumes a declaration of SQLCODE, and generates code to this effect.

Declaring SQLCA

Warning! Although SQLSTATE is preferred over SQLCODE and SQLCA, this version only fully supports SQLCODE. A future version will support SQLSTATE.

The syntax for declaring SQLCA is:

```
exec sql include sqlca;
```

You can use the Embedded SQL include statement to include other files in your application the same way you would use the C preprocessor `#include` command. You can also set a precompiler command option to specify an *include* file directory. At precompile time, the precompiler searches the path specified in the C compile command. The precompiler uses the *include* file path to search for this file. It opens and reads the included file as if were part of the main file. If the included file cannot be found, the precompile fails.

Multiple SQLCAs

You may have multiple SQLCAs, but each must follow C scoping rules for host variables. Each SQLCA need not be in a separate scope.

SQLCA variables

When the precompiler encounters the include `sqlca` statement, it inserts the SQLCA structure declaration into the application program. SQLCA is a data structure containing precompiler-determined *system variables*, each of which can be accessed independently. Your application program should never directly alter these variables.

SQLCA variables pass information to your application program about the status of the most recently executed Embedded SQL statement.

Table 3-1 describes the SQLCA variables that hold status information, return codes, error codes, and error messages generated by Adaptive Server Enterprise:

Table 3-1: Adaptive Server Enterprise SQLCA variables

Variable	Datatype	Description
<i>sqlcaid</i>	char	Text string that contains “sqlca”.
<i>sqlcabc</i>	long	Length of SQLCA.
<i>sqlcode</i>	long	Contains the return code of the most recently executed SQL statement. See SQLCODE values for return code definitions.
<i>sqlwarn[0]</i> to <i>sqlwarn[7]</i>	char	Warning flags. Each flag indicates whether a warning has been issued: a ‘W’ for warning, or a blank space for no warning. Chapter 8 describes the <i>sqlwarn</i> flags.
<i>sqlerrm.sqlerrmc</i> []	char	Error message.
<i>sqlerrm.sqlerrml</i>	long	Error message length.
<i>sqlerrp</i>	char	Procedure that detected error/warning.
<i>sqlerrd[6]</i>	long	Details of error/warning. [2] is the number of rows affected.

Accessing SQLCA variables

SQLCA variables are members of a C structure, `sqlca`, that is declared by the include `sqlca` statement. To access SQLCA variables, use the C structure member operator (`.`), as shown in the following example:

```
if (sqlca.sqlwarn[1] == 'W')
{
    printf("\nData truncated");
    return;
}
```

You can also pass the address of the sqlca structure to a function, then access the SQLCA variables within that function with the -> operator. The following example shows a function that works this way:

```
warning(p)
struct sqlca *p;
{
    if (p->sqlwarn[3] == 'W')
    {
        printf("\nIncorrect number of variables in
fetch.\n");
    }
    return;
}
```

SQLCA variables are useful for determining whether an Embedded SQL statement executed successfully. The other SQLCA variables listed in the previous section provide additional information about errors and return codes to help in debugging as well as the normal processing of your application.

SQLCODE within SQLCA

The application should test sqlcode after each statement executes, because Adaptive Server Enterprise updates it after each execution. As a rule, use the whenever statement, described in Chapter 8, “Handling Errors.” to perform this task.

Declaring SQLCODE as a standalone area

Warning! Although SQLSTATE is preferred over SQLCODE and SQLCA, this version only fully supports SQLCODE. A future version will fully support SQLSTATE.

As an alternative to creating a SQLCA, use SQLCODE independently. It contains the return code of the most recently executed SQL statement. The benefit of declaring SQLCODE as a standalone area is that it executes code faster. If you have no need to review the other information that SQLCA holds and are solely interested in return codes, consider using SQLCODE.

Despite SQLCODE's faster execution speed, SQLSTATE is preferred over SQLCODE, which is supported for its compatibility with earlier versions of Embedded SQL.

Note In a future version, you will be advised to use SQLSTATE instead of SQLCODE for receiving status results.

Following is an example of declaring SQLCODE as a standalone area:

```
long SQLCODE;
exec sql open cursor pub_id;
    while (SQLCODE == 0)
    {
        exec sql fetch pub_id into :pub_name;
```

For details on debugging any errors SQLCODE indicates, see Chapter 8, "Handling Errors."

Table 3-2 displays SQLCODE values:

Table 3-2: SQLCODE values

Value	Description
0	Statement executed successfully.
-n	Error occurred. See Server or Client-Library error messages. "-n" represents the number associated with the error or exception.
+100	No data exists, no rows left after fetch, or no rows met search condition for update, delete, or insert.

Using SQLSTATE

Warning! Although SQLSTATE is preferred over SQLCODE and SQLCA features, this version only fully supports SQLCODE. A future version will fully support both SQLCA and SQLSTATE.

SQLSTATE is a status parameter. Its codes indicate the status of the most recently attempted procedure—either the procedure completed successfully or an error occurred during the execution of the procedure.

SQLSTATE is a character-string parameter whose exceptions values are described in Table 3-3:

Table 3-3: SQLSTATE values

Value	Description
00XXX	Successful execution
01XXX	Warning
02XXX	No data exists; no rows affected
Any other value	Error

Obtaining SQLSTATE codes and error messages

SQLSTATE messages can be informational, warnings, severe, or fatal. Adaptive Server Enterprise and Open Client Client-Library generate the majority of SQLSTATE messages. See the appropriate documentation for a complete list of SQLSTATE codes and error messages.

See Appendix A, “Precompiler Warning and Error Messages,” for the table of SQLSTATE messages that the precompiler can generate.

Summary

This chapter explained SQLCA, SQLCODE, and SQLSTATE. After a statement executes, Adaptive Server Enterprise stores return codes and information in SQLCA variables, in a standalone SQLCODE area, or in SQLSTATE. These return codes indicate the failure or success of the statement that most recently executed.

Using Variables

This chapter details the following two types of variables that pass data between your application and Adaptive Server Enterprise:

- Host variables, which are C variables you use in Embedded SQL statements to hold data that is retrieved from and sent to Adaptive Server Enterprise
- Indicator variables, which you associate with host variables to indicate null data and data truncation

Topic	Page
Declaring variables	19
Using host variables	27
Using indicator variables	29
Using arrays	33
Scoping rules	34
Datatypes and Adaptive Server Enterprise	34

Declaring variables

As discussed in Chapter 3, the precompiler automatically sets the system variables when you include `SQLCA`, `SQLCODE`, or `SQLSTATE` in the application program. However, you must explicitly declare host and indicator variables in a `declare` section before using them in Embedded SQL statements.

Warning! The precompiler generates some variables, all of which begin with “_sql”. Do not begin your variables with “_sql”, or you may receive an error message or unreliable data.

The precompiler ignores macros and `#include` statements in a declare section. It processes include statements as if the contents of the included file were copied directly into the file being precompiled. The syntax for a declare section with an include statement is:

```
exec sql begin declare section;
    exec sql include "filename";
    ...
exec sql end declare section;
```

Host variable declarations must conform to the C rules for variable declarations. You need not declare all variables in one declare section, since you can have an unlimited number of declare sections in a program.

When you declare variables, you must also specify the **datatype**. See “Datatypes and Adaptive Server Enterprise” on page 34 for valid datatypes. Alternatively, use the Client-Library typedefs, such as `CS_CHAR`, which are declared in the *cspublic.h* file, in declare sections.

The following example shows two character strings defined in a declare section.

```
exec sql begin declare section;
    CS_CHAR name[20];
    CS_CHAR type[3];
exec sql end declare section;
```

When declaring a host variable, you can also initialize it but only if it is a scalar variable, such as this one:

```
exec sql begin declare section;
    int total = 0;
exec sql end declare section;
```

You cannot initialize an array in its declaration.

Using datatypes

In Embedded SQL, you can use the C datatypes `char`, `int`, `float`, `double`, and `void`. You can use the keywords `const` and `volatile`, though not with structures. You can use the keywords `unsigned`, `long`, and `short`. You can use storage class specifiers: `auto`, `extern`, `register`, and `static`.

Note Do not use long int when building 64-bit applications.

```
exec sql begin declare section;
```

```

register int frequently_used_host_variable;
extern char
shared_string_host_variable[STRING_SIZE];
/*
** The const restriction is not enforced by
** the precompiler; only the compiler makes use
** of it.
*/
const float
input_only_host_variable = 3.1415926;
/*
** Be careful. You can declare unsigned
** integers, but if you select a negative
** number into one, you will get an incorrect
** result and no error message.
*/
unsigned long int unsigned_host_variable;
exec sql end declare section;

```

You can declare pointers in the declare section, but you cannot use a pointer as a host variable in an Embedded SQL statement.

```

exec sql begin declare section;
int number;
/*
** It's convenient to declare this here,
** but we won't be using it as a host variable.
*/
int *next_number;
exec sql end declare section;

```

You can use the following Sybase datatypes:

```

CS_BINARY, CS_BIT, CS_BIGINT, CS_BOOL, CS_CHAR, CS_DATE,
CS_DATETIME, CS_DATETIME4, CS_DECIMAL, CS_FLOAT,
CS_REAL, CS_IMAGE, CS_INT, CS_MONEY, CS_MONEY4,
CS_NUMERIC, CS_RETCODE, CS_SMALLINT, CS_TEXT, CS_TIME
CS_TINYINT, CS_UBIGINT, CS_UINT, CS_UNICHAR, CS_UNITEXT,
CS_USMALLINT, CS_VOID, CS_XML.

```

CS_CHAR is treated differently from char; CS_CHAR is null-terminated but not blank-padded; char is null-terminated and blank-padded to the length of the array.

```

/*
** Your #define for the array size doesn't
** have to be in the declare section,
** though it would be legal if it were.

```

```
*/
#define MAX_NAME 40;

exec sql begin declare section;
    CS_MONEY salary;
    CS_CHAR print_this[MAX_NAME];
    char print_this_also[MAX_NAME];
exec sql end declare section;

exec sql select salary into :salary from salaries
    where employee_ID = '01234';
/*
** The CS_MONEY type is not directly printable.
** Here's an easy way to do a conversion.
*/
exec sql select :salary into :print_this;

/*
** This will not be blank-padded.
*/
printf("Salary for employee 01234 is %s.\n",
    print_this);

/*
** This will be blank-padded.
*/
exec sql select :salary into :print_this_also;
printf("Salary for employee 01234 is %s.\n",
    print_this_also);
```

Using type definitions

You can use a type definition (typedef) within a declare section to declare variables. For example:

```
exec sql begin declare section;
/*
** The typedef and the use of the typedef
** can be in separate declare sections
** if the typedef comes first.
** The typedef can even be in an "exec
** sql include file".
*/
typedef int STORE_ID;
```

```

        STORE_ID current_ID;
    exec sql end declare section;

    exec sql select store_ID into :current_ID
        from sales_table where
            store_name = 'Furniture Kingdom';

```

Type Definitions and Limits

Table 4-1 displays valid type definitions in Embedded SQL:

Table 4-1: Valid typedefs

Typedef	Description
CS_BINARY	Binary type
CS_BIT	Bit type
CS_CHAR	Character type
CS_DATE	Date type
CS_TIME	Time type
CS_DATETIME	Datetime type
CS_FLT8	8-byte float type
SQLINDICATOR	Used for indicator variables (2-byte integer)
CS_INT	4-byte integer
CS_BIGINT	8-byte integer
CS_MONEY	Money type
CS_SMALLINT	2-byte integer
CS_TINYINT	1-byte unsigned integer
CS_SMALLINT	2-byte integer
CS_USMALLINT	2-byte unsigned integer
CS_UINT	4-byte unsigned integer
CS_UBIGINT	8-byte unsigned integer
CS_TEXT	Text type
CS_IMAGE	Image type
CS_UNICHAR	UTF16 Unicode character type
CS_UNITEXT	UTF16 Unicode text type
CS_XML	xml data

All basic ANSI type definitions are also valid in Embedded SQL.

Implementation limits

The nesting depth for `exec sql include` *filename* limit is 32.

Using `#define`

You can use `#define` values in a declare section to dimension arrays and initialize variables. When you use `#define` in a host variable declaration, place it before the host variable declaration that uses it. For example, the following two examples are valid:

```
#define PLEN 26
CS_CHAR name[PLEN];
```

and:

```
exec sql begin declare section;
#define PLEN 26
exec sql end declare section;
...
exec sql begin declare section;
CS_CHAR name[PLEN];
exec sql end declare section;
```

You can use `#define` to declare *symbolic names*. Make the declaration before using it in the application. For example, to define “10” symbolically, use this nomenclature:

```
exec sql begin declare section;
#define count_1 10
CS_CHAR var1[count_1];
exec sql end declare section;
```

Declaring an array

The precompiler supports *complex definitions*, which are structures and arrays. You may nest structures, but you cannot have an **array** of structures.

The precompiler recognizes single-dimensional arrays of all datatypes.

The precompiler also recognizes double-dimensional arrays of characters, as the following example demonstrates:

```
#define maxrows 25
int numsales [maxrows];
exec sql begin declare section;
#define DATELEN 30
#define DAYS_PER_WEEK 7
CS_CHAR days_of_the_week[DAYS_PER_WEEK][DATELEN+1];
exec sql end declare section;
```


You can declare arrays of any datatype. However, to select into an array element, its datatype must be scalar—integer, character, floating point, or pointer. You can select into elements of any scalar array, even an array of structures, as shown:

```

exec sql begin declare section;
    int sales_totals[100];
    struct sales_record{
        int total_sales;
        char store_name[40];
    } sales_record[100];
exec sql end declare section;

/*
** If there are fewer than 100 stores,
** this will get the sales totals for all
** of them.  If there are more than
** 100, it will cause an error at runtime.
*/
exec sql select total_sales into :sales_totals
    from sales_table;
/*
** This gets the sales for just one store.
*/
exec sql select total_sales into :sales_totals[0]
    from sales_table where store_ID = 'xyz';
/*
** This gets two pieces of information on a single **
store.
*/
exec sql select total_sales, store_name
    into :sales_records[i]
    from sales_table where store_ID = 'abc';

```

Declaring character arrays

A character array can be of type `CS_CHAR` or `char[]`; however, the rules governing these two datatypes differ. When an array of type `char[]` is used as input, the precompiler checks that the array terminates with a null character. If the array is not null terminated, a precompiler runtime function returns an error. In contrast, an array of type `CS_CHAR` is not checked for null termination. Instead, the length of the input continues up to the null character, if present, or to the declared length of the array—whichever comes first.

When used as output, arrays of type `char[]` are padded with space characters (blank-padded) and null terminated. Arrays of type `CS_CHAR` are not blank padded, only null terminated.

A character array is scalar, because it represents a single string. Thus, you can select into an array of characters and get back just a single string. Also, unlike arrays of other datatypes, an array of characters can be a host input variable.

See “Using arrays” on page 33.

Declaring unions and structures

You can declare unions and structures, either directly or by using a type definition (typedef). You can use an element of a union as a host variable, but not the union as a whole. In contrast, a host variable can be either an entire structure or just one of the structure’s elements. The following example declares a union and a structure:

```
exec sql begin declare section;
    typedef int PAYMENT_METHOD;
    PAYMENT_METHOD method;
    union salary_or_percentage {
        CS_MONEY salary;
        CS_NUMERIC percentage;
    } amount;
    struct employee_record {
        char first_name[30];
        char last_name[30];
        char employee_ID[30];
    } this_employee;
    char *employee_of_the_month_ID = "01234567";
exec sql end declare section;

exec sql select first_name, last_name, employee_ID
    into :this_employee
    from employee_table
    where employee_ID = :employee_of_the_month_ID;
exec sql select payment_type into :method
    from remuneration_table where employee_ID =
    :this_employee.employee_ID;
switch (method) {
    case SALARIED:
        exec sql select salary into
            :amount.salary
            from remuneration_table
```

```

        where employee_ID =
            this_employee.employee_ID;
        break;
    case VOLUNTEER:
        exec sql select 0 into
            :amount.salary
        break;
    case COMMISSION:
        exec sql select commission_percentage into
            :amount.percentage
        from remuneration_table
        where employee_ID =
            this_employee.employee_ID;
        break;
    }

```

Using host variables

Host variables let you transfer values between Adaptive Server Enterprise and the application program.

Declare the host variable within the application program's Embedded SQL declare section. Only then can you use the variable in SQL statements.

When you use the variable within an Embedded SQL statement, prefix the host variable with a colon. When you use the variable elsewhere in the program, do not use a colon. When you use several host variables successively in an Embedded SQL statement, separate them with commas or follow the grammar rules of the SQL statement.

The following example demonstrates how to use a variable. *user* is defined in a declare section as a character variable. Then, it is used as a host variable in a select statement:

```

exec sql begin declare section;
    CS_CHAR user[32];
exec sql end declare section;

exec sql select user_name() into :user;
printf("You are logged in as %s.\n", user);

```

There are four ways to use host variables. Use them as:

- Input variables for SQL statements and procedures

- Result variables
- Status variables from calls to SQL procedures
- Output variables for SQL statements and procedures

Declare all host variables as described in “Declaring variables” on page 19, regardless of their function. Following are instructions for using host variables.

Host input variables

These variables pass information to Adaptive Server Enterprise. The application program assigns values to them. They hold data used in executable statements such as stored procedures, select statements with where clauses, insert statements with values clauses, and update statements with set clauses.

The following example uses the variables *id* and *publisher* as input variables:

```
exec sql begin declare section;
    CS_CHAR id[7];
    CS_CHAR publisher[5];
exec sql end declare section;
...
exec sql delete from titles where title_id = :id;
exec sql update titles set pub_id = :publisher
    where title_id = :id;
```

Host result variables

These variables receive the results of select and fetch statements.

The following example uses the variable *id* as a **result variable**:

```
exec sql begin declare section;
    CS_CHAR id[5];
exec sql end declare section;

exec sql select title_id into :id from titles
    where pub_id = "0736" and type = "business";
```

Host status variables

These variables receive the return status values of stored procedures. Status variables indicate whether the stored procedure completed successfully or the reasons it failed.

Declare status variables as 2-byte integers (CS_SMALLINT).

The following example uses the variable *retcode* as a **status variable**:

```
exec sql begin declare section;
  CS_SMALLINT  retcode;
exec sql end declare section;

exec sql begin transaction;
exec sql exec :retcode = update_proc;
if (retcode != 0)
{
  exec sql rollback transaction;
}
```

Host output variables

These variables pass data from stored procedures to the application program. Use host output variables when stored procedures return the value of parameters declared as out.

The following example uses the variables *par1* and *par2* as output variables:

```
exec sql exec a_proc :par1 out, :par2 out;
```

Using indicator variables

You can associate indicator variables with host variables to indicate when a database value is null. Use a space and, optionally, the indicator keyword, to separate each indicator variable from the host variable with which it is associated. Each **indicator variable** must immediately follow its host variable.

Without indicator variables, Embedded SQL cannot indicate null values.

Indicator variables and server restrictions

Embedded SQL is a generic interface that can run on a variety of servers, including Adaptive Server Enterprise.

Because it is generic, Embedded SQL does not enforce or reflect any particular server's restrictions.

When writing an Embedded SQL application, keep the application's ultimate target **server** in mind. If you are unsure about what is legal on a server and what is not, consult your server documentation.

Using host variables with indicator variables

Declare host and indicator variables in a declare section before using them anywhere in an application program containing Embedded SQL statements. Declare indicator variables as 2-byte integers (short or CS_SMALLINT) in a declare section before using them.

Prefix indicator variables with a colon when using them in an Embedded SQL statement.

The syntax for associating an indicator variable with a host variable is:

```
:host_variable [[indicator] :indicator_variable]
```

The association between an indicator and host variable lasts only for the duration of a statement— that is, for the duration of one `exec sql` statement, or between open and close cursor statements. A value is assigned to the indicator variable at the same time a value is assigned to the host variable.

Adaptive Server Enterprise sets the indicator variable only when you assign a value to the host variable. Therefore, you can declare an indicator variable once and reuse it with different host variables in different statements.

You can use indicator variables with output, result, and input variables. When used with output and result variables, Embedded SQL sets the variable to indicate the null status of the associated host variable. When used with input variables, you set the value of the indicator variable to show the null status of the **input variable** before submitting it to Adaptive Server Enterprise.

Using indicator variables with host output and result variables

When you associate an indicator variable with an output or result variable, Client-Library automatically sets it to one of the following values in Table 4-2:

Table 4-2: Indicator variable values when used with output or result variable

Value	Meaning
-1	The corresponding database column in Adaptive Server Enterprise contains a null value.
0	A non-null value was assigned to the host variable.
>0	An overflow occurred while data was being converted for the host variable. The host variable contains truncated data. The positive number represents the length, in bytes, of the value before it was truncated.

The following example demonstrates associating the indicator variable *indic* with the result variable *id*:

```
exec sql begin declare section;
  CS_CHAR          id[6];
  CS_SMALLINT      indic;
  CS_CHAR          pub_name[41];
exec sql end declare section;

exec sql select pub_id into :id indicator :indic
  from titles where title
  like "%Stress%";

if (indic == -1)
{
  printf("\n pub_id is null");
}
else
{
  exec sql select pub_name into :pub_name
    from publishers where pub_id = :id;
  printf("\nPublisher: %s", pub_name);
}
```

Using indicator variables with host input variables

When you associate an indicator variable with an input variable, you must explicitly set the indicator variable, using the values in Table 4-3 as a guide.

Table 4-3: Indicator variable values used with input variable

Value	Meaning
-1	Treat the corresponding input as a null value.
0	Assign the value of the host variable to the column.

You must supply host language code to test for a null input value and set the indicator variable to -1. This informs Client-Library of a null value. When you set the indicator variable to -1, null is used regardless of the host variable's actual value.

The following example demonstrates associating an indicator variable with an input variable. The database royalty column is set to a null value because *indic* is set to -1. Changing the value of *indic* changes the value of *royalty*.

```
exec sql begin declare section;
  CS_SMALLINT      indic;
  CS_INT           royalty;
exec sql end declare section;

indic = -1;
exec sql update titles set royalty = :royalty
      :indic where pub_id = "0736";
```

Host variable conventions

A **host variable** name must conform to C naming conventions.

You can use a host variable in an Embedded SQL statement wherever a Transact-SQL literal can be used in a Transact-SQL statement at the same location.

A host variable must conform to the valid precompiler datatypes. The datatype of a host variable must be compatible with the datatype of the database column values returned. See Table 4-5 on page 37 and Table 4-6 on page 38 for details. You cannot use host language reserved words and Embedded SQL keywords as variable names.

A host variable cannot represent Embedded SQL keywords or database objects, except as specified in **dynamic SQL**. See Chapter 7, "Using Dynamic SQL."

When a host variable represents a character string in a SQL statement, do not place it within quotes.

The following example is invalid because the precompiler inserts quotes around values when necessary. You should not type the quotes.

```
strcpy (p_id, "12345");
exec sql select pub_id into :p_id from publishers
where pub_id like ":p_id";
```

The following example is valid:

```
strcpy (p_id, "12345");
exec sql select pub_id into :p_id from publishers
where pub_id like :p_id;
```

Using arrays

An array is a group of related pieces of data associated with one variable. You can use arrays as output variables for the into clause of select and fetch statements. For example:

```
exec sql begin declare section;
  CS_CHAR au_array [100] [30];
exec sql end declare section;
exec sql
  select au_lname
  into :au_array
  from authors;
```

Note You can fetch a single item anywhere into an array. However, you can fetch multiple rows only into the beginning of an array.

For details on using arrays with select and fetch into, see “Selecting multiple rows using arrays” on page 49.

Multiple arrays

When you use multiple arrays within a single SQL statement, they should be the same size. Otherwise, you will receive an error message.

Scoping rules

The precompiler supports the C programming rules for variable scoping. Host variables defined within nested programs can use the external clause plus the variable name. For example:

```
FILE 1:
CS_CHAR  username [31]
main()
{
    sub1();
    printf("%s\n", username);
}
FILE 2:
void sub1()
{
    exec sql begin declare section;
    extern char username[31];
    exec sql end declare section;
    exec sql select USER() into :username;
    return;
}
```

Datatypes and Adaptive Server Enterprise

Host variable datatypes must be compatible with the datatypes of the corresponding database columns. So, before writing your application program, check the datatypes of the database columns. To ensure that your host variables are compatible with the Adaptive Server Enterprise datatypes, use the Sybase-supplied type definitions.

Table 4-4 shows and briefly describes the equivalent datatypes. For detailed descriptions of each Adaptive Server Enterprise datatype, see the *Adaptive Server Enterprise Reference Manual*.

Table 4-4: Comparison of C and Adaptive Server Enterprise-compatible datatypes

Sybase-supplied typedef	Description	C datatype	Adaptive Server Enterprise datatype
CS_BIGINT	8-byte integer type	long long	bigint
CS_BINARY	Binary type	unsigned char	binary, varbinary
CS_BIT	Bit type	unsigned char	boolean
CS_CHAR	Character type	char[n]	char, varchar
CS_DATE	4-byte date type	None	date
CS_TIME	4-byte time type	None	time
CS_DATETIME	8-byte datetime type	None	datetime
CS_DATETIME4	4-byte datetime type	None	smalldatetime
CS_BIGDATETIME	8-byte binary type	None	bigdatetime
CS_BIGTIME	8-byte binary type	None	bigtime
CS_TINYINT	1-byte unsigned integer type	unsigned char	tinyint
CS_SMALLINT	2-byte integer type	short	smallint
CS_INT	4-byte integer type	long	int
CS_DECIMAL	Decimal type	None	decimal
CS_NUMERIC	Numeric type	None	numeric
CS_FLOAT	8-byte float type	double	float
CS_REAL	4-byte float type	float	real
CS_MONEY	8-byte money type	None	money
CS_MONEY4	4-byte money type	None	smallmoney
CS_TEXT	Text type -y option required	unsigned char	text
CS_IMAGE	Image type -y option required	unsigned char	image
CS_UBIGINT	8-byte unsigned integer type	unsigned long long	ubigint
CS_UINT	4-byte unsigned integer type	unsigned int	uint
CS_UNICHAR	2-byte UTF-16 Unicode character type	unsigned short	unichar

Sybase-supplied typedef	Description	C datatype	Adaptive Server Enterprise datatype
CS_UNITEXT	2-byte UTF-16 Unicode text type	unsigned short	unitext
CS_USMALLINT	2-byte unsigned integer type	unsigned short	usmallint
CS_XML	XML type	unsigned char	xml

Converting datatypes

The precompiler automatically compares the datatypes of host variables with the datatypes of table columns in Adaptive Server Enterprise. If the Adaptive Server Enterprise datatype and the host language datatype are compatible but not identical, the precompiler converts one type to the other. Datatypes are compatible if the precompiler can convert the data from one type to the other. If the datatypes are incompatible, a conversion error occurs at runtime and sqlcode is set to <0.

Be careful when converting a longer datatype into a shorter one, such as a 4-byte into 2-byte, because there is always a possibility of truncating data. If a truncation occurs, sqlwarn1 is set to "W."

Converting datatypes for result variables

Table 4-5 shows which data conversions are valid for result variables. A bullet indicates that conversion is possible, but be aware that certain types of errors can result if you are not careful when choosing host variable datatypes.

Table 4-5: Datatype conversions for result variables

From: Adaptive Server Enterprise datatype	To: Sybase datatype definition															
	CS_TINYINT	CS_SMALLINT	CS_USMALLINT	CS_INT	CS_UINT	CS_BIGINT	CS_UBIGINT	CS_REAL	CS_CHAR	CS_UNICHAR	CS_MONEY	CS_DATE	CS_TIME	CS_DATETIME	CS_TEXT	CS_XML
char	•	•	•	•	•	•	•	•	•	•	•			•	•	•
unichar	•	•	•	•	•	•	•	•	•	•	•			•	•	•
varchar	•	•	•	•	•	•	•	•	•	•	•			•	•	•
bit	•	•	•	•	•	•	•	•	•	•	•				•	•
binary	•	•	•	•	•	•	•	•	•	•	•				•	•
tinyint	•	•	•	•	•	•	•	•	•	•	•				•	•
smallint	•	•	•	•	•	•	•	•	•	•	•				•	•
int	•	•	•	•	•	•	•	•	•	•	•				•	•
bigint	•	•	•	•	•	•	•	•	•	•	•				•	•
ubigint	•	•	•	•	•	•	•	•	•	•	•				•	•
uint	•	•	•	•	•	•	•	•	•	•	•				•	•
usmallint	•	•	•	•	•	•	•	•	•	•	•				•	•
float	•	•	•	•	•	•	•	•	•	•	•				•	•
money	•	•	•	•	•	•	•	•	•	•	•				•	•
date									•	•		•				
time									•	•			•			
datetime									•	•				•		
decimal	•	•	•	•	•	•	•	•	•	•	•				•	•
numeric	•	•	•	•	•	•	•	•	•	•	•				•	•
text	•	•	•	•	•	•	•	•	•	•	•				•	•
xml	•	•	•	•	•	•	•	•	•	•	•				•	•

Converting datatypes for input variables

Table 4-6 shows valid data conversions for input variables. A bullet indicates that conversion is possible; an "X" indicates that conversion is required. Errors, including truncation, can result if you choose nonconvertible host variable datatypes.

Table 4-6: Datatype conversions for input variables

To: Adaptive Server Enterprise datatype

From: C datatype	tinyint	bit	smallint	usmallint	int	uint	bigint	ubigint	float	char	unichar	money	date	time	datetime	decimal	numeric	text	xml
unsigned char	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
unichar	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
short int	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
long int	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
bigint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
ubigint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
uint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
usmallint	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
double float	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
char	X	X	X	X	X	X	X	X	X	•	•	X	•	•	•	X	X	X	X
money	•	•	•	•	•	•	•	•	•	•	•	•				•	•	•	•
date													•						
time														•					
datetime										X	X				•				
text	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•
xml	•	•	•	•	•	•	•	•	•	X	X	•				•	•	•	•

X – indicates that an explicit conversion is required.

Connecting to Adaptive Server Enterprise

This chapter explains how to connect an Embedded SQL program to Adaptive Server Enterprise and describes how to specify servers, user names, and passwords.

Topic	Page
Connecting to a server	39
Changing the current connection	41
Establishing multiple connections	41
Disconnecting from a server	44

Connecting to a server

A connection enables an Embedded SQL program to access a database and perform SQL operations.

Use the connect statement to establish a connection between an application program and Adaptive Server Enterprise. If an application uses both C and COBOL languages, the first connect statement must be issued from a COBOL program. See *Open Client Embedded SQL/COBOL Programmers Guide* for information.

The syntax for the connect statement is:

```
exec sql connect :user [identified by :password]
                [at :connection_name] [using :server]
```

Each of the following sections describes one of the connect statement's arguments. Only the *user* argument is required for the connect statement. The other arguments are optional.

user

user is a host variable or quoted string that represents an Adaptive Server Enterprise user name. The user name must be valid for the server specified.

password

password is a host variable or quoted string that represents the password associated with the specified user name. This argument is necessary only if a password is required to access Adaptive Server Enterprise. If the password argument is null, the user does not need to supply a password.

connection_name

connection_name uniquely identifies the Adaptive Server Enterprise connection. It can be a quoted literal. You can create an unlimited number of connections in an application program, one of which can be unnamed. *connection_name* has a maximum size of 255 characters.

When you use *connection_name* in a connect statement, all subsequent Embedded SQL statements that specify the same connection automatically use the server indicated in the connect statement. If the connect statement specifies no server, the default server is used. See the *Open Client and Open Server Programmers Supplement* for details on how the default server is determined.

Note To change the current server connection, use the set connection statement described in “Changing the current connection” on page 41.

An Embedded SQL statement should only reference a *connection_name* specified in a connect statement. At least one connect is required for each server that the application program uses.

server

server is a host variable or quoted string that represents a server name. *server* must be a character string that uniquely and completely identifies a server.

connect example

The following example connects to the server SYBASE using the password “passes.”

```
exec sql begin declare section;
CS_CHAR user[16];
CS_CHAR passwd[16];
CS_CHAR server[BUFSIZ];
exec sql end declare section;

strcpy(server, "SYBASE");
strcpy(passwd, "passes");
strcpy(user, "my_id");

exec sql connect :user identified by :passwd using
:server;
```

Changing the current connection

Use the set connection statement to change the current connection. The statement’s syntax is as follows:

```
exec sql set connection {connection_name | default}
```

where default is the unnamed connection, if any.

The following example changes the current connection:

```
exec sql connect "ME" at connect1 using "SERVER1";
exec sql connect "ME" at connect2 using "SERVER2";
exec sql set connection connect1;
exec sql select user_id() into :myid;
```

Establishing multiple connections

Some Embedded SQL applications require or benefit from having more than one active Adaptive Server Enterprise connection. For example:

- An application that requires multiple Adaptive Server Enterprise login names can have a connection for each login account.

- By connecting to more than one server, an application can simultaneously access data stored on different servers.

A single application can have multiple connections to a single server or multiple connections to different servers. Use the connect statement's *atconnection_name* clause to name additional connections for an application.

If you open a connection and then another new named or unnamed connection, the new connection is the current connection.

Note If you are generating stored procedures with the precompiler for appropriate SQL statements, then for each Embedded SQL file, the precompiler generates a single file for all stored procedures on all servers. You can load this file into the appropriate server(s). Although the server(s) will report warnings and errors about being unable to read the procedures intended for other servers, ignore them. The stored procedures appropriate for each server will load properly on that server. Be sure to load the stored procedures on all applicable servers or your queries fail.

Naming a connection

Table 5-1 shows how a connection is named:

Table 5-1: How a connection is named

If this clause is used	But without	Then, the connection name is
<i>at connection_name</i>		<i>connection_name</i>
using <i>server_name</i>	at	<i>server_name</i>
None		Actual name of the "DEFAULT" connection

Invalid statements with the *at* clause

The following statements are invalid with the *at* clause:

- connect
- begin declare section
- end declare section
- include file

- include sqlca
- set connection
- whenever

Using Adaptive Server Enterprise connections

Specify a connection name for any Embedded SQL statement that you want to execute on a connection other than the default unnamed connection. If your application program uses only one connection, you can leave the connection unnamed. Then, you do not need to use the at clause.

The syntax for using multiple connections is:

```
exec sql [at connection_name] sql_statement;
```

where `sql_statement` is a Transact-SQL statement.

The following example shows how two connections can be established to different servers and used in consecutive statements:

```
...

exec sql begin declare section;
CS_CHAR user[16];
CS_CHAR passwd[16];
CS_CHAR name;
CS_INT value, test;
CS_CHAR server_1[BUFSIZ];
CS_CHAR server_2[BUFSIZ];
exec sql end declare section;
...
strcpy (server_1, "sybase1");
strcpy (server_2, "sybase2");
strcpy(user, "my_id");
strcpy(passwd, "mypass");

exec sql connect :user identified by :passwd
at connection_2 using :server_2;

exec sql connect :user identified by :passwd using
:server_1;

/* This statement uses the current "server_1"
connection */
exec sql select royalty into :value from authors
```

```
where author = :name;

if (value == test)
{
/* This statement uses connection "connection_2" */
exec sql at connection_2 update authors
set column = :value*2
where author = :name;
}
```

Disconnecting from a server

The connections your application program establishes remain open until you explicitly close them or your program terminates. Use the `disconnect` statement to close a connection between the application program and Adaptive Server Enterprise.

The statement's syntax is as follows:

```
exec sql disconnect {connection_name | current | DEFAULT
| all}
```

where:

- `current` specifies the current connection.
- `DEFAULT` specifies the unnamed default connection.
- `all` specifies all connections currently in use.

The `disconnect` statement performs the following, in order:

- 1 Rolls back the current transactions ignoring any established savepoints.
- 2 Closes the connection.
- 3 Drops all temporary objects, such as tables.
- 4 Closes all open cursors.
- 5 Releases locks established for the current transactions.
- 6 Terminates access to the server's databases.

disconnect does not implicitly commit current transactions.

Warning! Before the program exits, make sure you perform an `exec sql disconnect` or `exec sql disconnect all` statement for each open connection. In some configurations, Adaptive Server Enterprise may not be notified when a **client** exits without disconnecting. If this happens, resources held by the application will not be released.

This chapter explains how to use Transact-SQL statements with Embedded SQL and host variables. It also explains how to use *stored procedures*, which are collections of SQL statements stored in Adaptive Server Enterprise. Because stored procedures are compiled and saved in the **database**, they execute quickly without being recompiled each time you invoke them.

Topic	Page
Transact-SQL statements in Embedded SQL	47
Selecting rows	48
Grouping statements	69

Transact-SQL statements in Embedded SQL

The following sections identify the Transact-SQL statements and their differences in Embedded SQL.

exec sql syntax

Embedded SQL statements must begin with the keywords `exec sql`. The syntax for Embedded SQL statements is:

```
exec sql [at connection_name] sql_statement
```

where:

- *connection_name* specifies the connection for the statement. See Chapter 5, “Connecting to Adaptive Server Enterprise,” for a description of connections. The `at` keyword is valid for Transact-SQL statements and the `disconnect` statement.
- *sql_statement* is one or more Transact-SQL statements.

Invalid statements

Except for the following Transact-SQL statements, all Transact-SQL statements are valid in Embedded SQL:

- print
- readtext
- writetext

Transact-SQL statements that differ in Embedded SQL

While most Transact-SQL statements retain their functionality and syntax when used in Embedded SQL, the select, update, and delete statements (the Data Manipulation Language, or DML, statements) can be slightly different in Embedded SQL:

- The following items are specific to the into clause of the select statement:
 - The into clause can assign one row of data to scalar host variables. This clause is valid only for select statements that return just one row of data. If you select multiple rows, a negative SQLCODE results, and only the first row is returned.
 - If the variables in an into clause are arrays, you can select multiple rows. If you select more rows than the array holds, an exception of SQLCODE <0 is raised, and the extra rows are lost.
 - select cannot return multiple rows of data in host variables, except through a cursor or by selecting into an array.
- The update and delete statements can use the search condition where current of *cursor_name*.

Selecting rows

There can be a maximum of 1024 columns in a select statement. For the complete listing of the select statement's syntax, see the *Adaptive Server Enterprise Reference Manual*.

Selecting one row

When you use the select statement without a cursor or array, it can return only one row of data. Embedded SQL requires a cursor or an array to return more than one row of data.

In Embedded SQL, a select statement must have an into clause. The clause specifies a list of host variables to be assigned values.

Note The current Embedded SQL precompiler version does not support into clauses that specify tables.

The syntax of the Embedded SQL select statement is:

```
exec sql [at connect_name ]
        select [all | distinct] select_list into
           :host_variable[[indicator]:indicator_variable]
           [, :host_variable
           [[indicator]:indicator_variable]...];
```

For additional information on select statement clauses, see the *Adaptive Server Enterprise Reference Manual*.

The following select statement example accesses the authors table in the pubs2 database and assigns the value of au_id to the host variable *id*:

```
exec sql select au_id into :id from authors
        where au_lname = "Stringer";
```

Selecting multiple rows using arrays

You can return multiple rows with arrays. The two array actions involve selecting and fetching into arrays.

select into arrays

Use the select into array method when you know the maximum number of rows that will be returned. If a select into statement attempts to return more rows than the array can hold, the statement returns the maximum number of rows that the smallest array can hold.

Example

Following is an example of selecting into an array:

```
exec sql begin declare section;
        CS_CHAR titleid_array [100] [6];
```

```
exec sql end declare section;
...
exec sql select title_id into :titleid_array
      from titles;
```

Indicators with array fetches

To use indicators with array fetches, declare an array of indicators of the same length as the *host_variable* array, and use the syntax for associating the indicator with the host variable.

Example

Following is an example of using indicators with array fetches:

```
exec sql begin declare section;
      int item_numbers [100];
      short i_item_numbers [100];
exec sql end declare section;
...
exec sql select it_n from item.info
      into :item_numbers :i_item_numbers;
...

```

Arrays and structures as indicator variables

For tables with a large number of columns you can use arrays and structures as a set of host variables that is referenced in a SQL statement. An indicator variable is always a 2-byte integer (short).

Examples

Example 1 This is an example of declaring indicator arrays:

```
EXEC SQL BEGIN DECLARE SECTION;

/* Destination variables for fetches, using an */
/* array.*/
struct _hostvar {
      int m_titleid;
      char m_title[65];
      char m_pubname[41];
      char m_pubcity[21];
      char m_pubstate[3];
      char m_notes[201];
      float m_purchase;
} host_var1;

/* An indicator array for all variables. */
short indic_var[7];
```

```
EXEC SQL END DECLARE SECTION;
```

Example 2 This is an example of declaring indicator structures:

```
EXEC SQL BEGIN DECLARE SECTION;
/* Destination variables for fetches, using a */
/* struct.*/
struct _hostvar {
    int m_titleid;
    char m_title[65];
    char m_pubname[41];
    char m_pubcity[21];
    char m_pubstate[3];
    char m_notes[201];
    float m_purchase;
} host_var1;

/* An indicator structure for above variables. */
struct _indicvar {
    short i_titleid;
    short i_title;
    short i_pubname;
    short i_pubcity;
    short i_pubstate;
    short i_notes;
    short i_purchase;
} indic_var1;

EXEC SQL END DECLARE SECTION;
```

Example 3 This is an example of executing a query on indicator arrays or indicator structures:

```
EXEC SQL
SELECT titleid, title, pubname, city, state, notes,
       purchases
INTO :host_var1 INDICATOR :indic_var1
FROM T1, T2
WHERE ...
```

Usage

When using structs and arrays as indicator variables:

- The number of elements in the indicator array or struct must be exactly the same as the number of elements in the host variable structure. A mismatch causes `cpre` or `cpre64` to stop processing, and code is not generated.
- The columns in the `SELECT` list must match by sequence, and datatype, the chosen structure name in the `INTO` list. A mismatch causes `ct_bind()` runtime errors and stops processing.

- INDICATOR is an optional keyword, and can be omitted. However, the “:” before the hostvar struct and the indicator array or struct is required.

Error messages

Table 6-1 describes the Embedded SQL internal error messages created to handle host variable versus indicator variable mismatch errors for this feature.

Table 6-1: New internal error messages

Message ID	Message text	Severity	Fix
M_INVTYPE_V	Incorrect type of indicator variable found in the structure.	Fatal	Make sure that the same indicator variable is used in the hostvar and indicator declarations.
M_INVTYPE_VI	Mismatch between number of structure elements in the indicator structure and hostvar structure.	Fatal	Declare the same number of elements in the indicator structure and hostvar structure.
M_INVTYPE_VII	Mismatch between number of elements in the indicator array and hostvar structure.	Fatal	Declare the same number of elements in the indicator array and hostvar structure.

Limitation

You cannot mix singleton host variables or singleton indicator variables with hostvar structures, and indicator arrays or structures.

fetch into batch arrays

fetch returns the specified number of rows from the currently active set. Each fetch returns the subsequent batch of rows. For example, if the currently active set has 150 rows and you select and fetch 60 rows, the first fetch returns the first 60 rows. The next fetch returns the following 60 rows. The third fetch returns the last 30 rows.

Note To find the total number of rows fetched, see the *SQLERRD* variable in the *SQLCA*, as described in “SQLCA variables” on page 15.

Cursors and arrays

Use the fetch into array method when you do not know the number of rows to be returned into the array. Declare and open a cursor, then use fetch to retrieve *groups of rows*. If a fetch into attempts to return more rows than the array can hold, the statement returns the maximum number of rows that the smallest array can hold and *SQLCODE* displays a negative value, indicating that an error or exception occurred.

Selecting multiple rows using cursors

You can also use cursors to return multiple rows. A **cursor** is a data selector that passes multiple rows of data to the host program, one row at a time. The cursor indicates the first row, also called the **current row**, of data and passes it to the host program. With the next fetch statement, the cursor advances to the next row, which has now become the current row. This continues until all requested rows are passed to the host program.

Use a cursor when a select statement returns more than one row of data. Client-Library tracks the rows Adaptive Server Enterprise returns and buffers them for the application. To retrieve data with a cursor, use the fetch statement.

The cursor mechanism is composed of these statements:

- declare
- open
- fetch
- update and delete where current of
- close

Cursor scoping rules

The rules that govern the initial scope of a cursor differ, depending on whether the cursor is static or dynamic. However, after a static cursor is opened or a dynamic cursor is declared, the scoping rules for both types of cursors are the same. The rules are as follows:

- Until a static cursor is open, its scope is limited to the file where the cursor was declared. Any statement that opens the static cursor must be in this file. After a static cursor is open, its scope is limited to the connection on which the cursor was opened.
- A dynamic cursor does not exist until it is declared. After it is declared, its scope is limited to the connection on which it was declared.
- A cursor name can be open on more than one connection at a time.
- Statements that fetch, update, delete, or close a cursor can appear in files other than the one where the cursor is declared. Such statements, however, must execute on the connection where the cursor was opened.

Identically named static cursors

- If you declare a static cursor in an Embedded SQL/C, open the cursor, use it to fetch data, close the cursor, and then do not deallocate it, subsequently declaring a cursor with the same name and DML will not result in an error. At the second declaration, the Embedded SQL/C program simply notices that a cursor of the same name and DML already exists, disregards the second declaration, and reopens the existing cursor. However, if you redeclare a cursor of the same name but with different DML, you may receive an error. Also, if you attempt to open an existing static cursor that has not been closed, you receive the following error:

```
SQLCODE= (-16843032)
Adaptive Server Error
ct_cursor(OPEN): user api layer: external error: The
cursor on this command structure has already been
opened.
```

- If you declare a static cursor using the isql utility, open the cursor, use it to fetch data, close the cursor, and then do not deallocate it, subsequently declaring a cursor with the same name and DML results in an isql error. In isql, you must deallocate the existing cursor before you can redeclare it.

Identically named dynamic cursors

- If you declare a dynamic cursor in an Embedded SQL/C program or using the isql utility, open the cursor, use it to fetch data, close the cursor, and then do not deallocate it, subsequently declaring a cursor with the same name and DML will result in an error. In an Embedded SQL/C program or in isql, you must first deallocate the existing dynamic cursor before you can redeclare it. If you attempt to declare a dynamic cursor that has already been declared and not yet deallocated, you receive the following error:

```
SQLCODE= (-16843030)
Adaptive Server Error
ct_dynamic(CURSOR_DECLARE): user api layer: external
error: A cursor has already been declared on this
command structure.
```

Declaring cursors

Declare a cursor for each select statement that returns multiple rows of data. You must declare the cursor before using it, and you cannot declare it within a declare section.

Note The declare cursor statement is a declaration, not an executable statement. Therefore, it may appear anywhere in a file; SQLCODE, SQLSTATE, and SQLCA are not set after this statement.

The syntax for declaring a cursor is:

```
exec sql declare cursor_name cursor
        for select_statement ;
```

where:

- *cursor_name* identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols “#” or “_”.
- *select_statement* is a select statement that can return multiple rows of data. The syntax for select is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use into or compute clauses.

Example

The following example demonstrates declaring cursors:

```
exec sql declare c1 cursor for
        select type, price from titles
        where type like :wk-type;
```

In this example, *c1* is declared as a cursor for the rows that will be returned for the type and price columns. The precompiler generates no code for the declare cursor statement. It simply stores the select statement associated with the cursor.

When the cursor opens, the select statement or procedure in the declare cursor statement executes. When the data is fetched, the results are copied to the host variables.

Note Each cursor’s open and declare statements must be in the same file. Host variables used within the declare statement must have the same scope as the one in which the open statement is defined. However, once the cursor is open, you can perform fetch and update or delete where current of on the cursor in any file.

Declaring scrollable cursors

The syntax for declaring a scrollable cursor is:

```
exec sql declare  cursor_name  [cursor sensitivity]
                 [cursor scrollability] cursor
                 for  select_statement ;
```

where:

- *cursor_name* identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols “#” or “_”.
- *cursor sensitivity* specifies the sensitivity of the cursor. The options are:
 - *semi_sensitive*. If *semi_sensitive* is specified in the declare statement, scrollability is implied. The cursor is *semi_sensitive*, scrollable, and read-only.
 - *insensitive*. If *insensitive* is specified in the declare statement, the cursor is *insensitive*. Scrollability is determined by specifying **SCROLL** in the declare part. If **SCROLL** is omitted or **NOSCROLL** is specified, the cursor is *insensitive* only and non-scrollable. It is also read-only.

If cursor sensitivity is not specified, the cursor is non-scrollable and read-only.

- *cursor scrollability* specifies the scrollability of the cursor. The options are:
 - *scroll*. If *scroll* is specified in the declare statement and sensitivity is not specified, the cursor is *insensitive* and scrollable. It is also read-only.
 - *no scroll*. If the **SCROLL** option is omitted or **NOSCROLL** is specified, the cursor is non-scrollable and read-only. See the previous *cursor sensitivity* description for cursor behavior.

If cursor scrollability is not specified, the cursor is non-scrollable and read-only.

- *select_statement* is a select statement that can return multiple rows of data. The syntax for *select* is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use *into* or *compute* clauses.

Declaring cursors that release locks at cursor close

The syntax for declaring cursors that release locks when the cursor closes is:

```
exec sql declare  cursor_name  [cursor sensitivity]
                 [cursor scrollability] [release_locks_on_close]
                 cursor for  select_statement
                 [for {read only | update [ of column_name_list]}] ;
```

where:

- *cursor_name* identifies the cursor. The name must be unique and have a maximum of 255 characters. The name must begin with a letter of the alphabet or with the symbols “#” or “_”.
- *cursor sensitivity* – See “Declaring scrollable cursors” on page 56.
- *cursor scrollability* – See “Declaring scrollable cursors” on page 56.
- *select_statement* is a select statement that can return multiple rows of data. The syntax for select is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use into or compute clauses.
- *column_name_list* identifies the list of columns to be affected.

You cannot use *release_locks_on_close* with an update clause except in this form:

```
exec sql declare cursor c1 release_locks_on_close
                 cursor for select * from T for update of col_a
```

In this case, *release_locks_on_close* is ignored.

Note cpre cannot generate these *ct_cursor()* options:

- CS_CUR_RELLOCKS_ONCLOSE | CS_READ_ONLY
 - CS_CUR_RELLOCKS_ONCLOSE | CS_FOR_UPDATE
-

ESQL/C sample code is available in *example8.cp*.

Opening cursors

To retrieve the contents of selected rows, you must first open the cursor. The open statement executes the select statement associated with the cursor in the declare statement.

The open statement’s syntax for opening a cursor is:

```
exec sql open cursor_name;
```

The open statement's syntax for opening a scrollable cursor is:

```
exec sql open cursor_name [ROW_COUNT = size];
```

Note ROW_COUNT should only be specified with (scrollable) cursors when arrays are used as host variables.

After you declare a cursor, you can open it wherever you can issue a select statement. When the open statement executes, Embedded SQL substitutes the values of any host variables referenced in the declare cursor statement's where clause.

The number of cursors you may have open depends on the resource demands of the current session. Adaptive Server Enterprise does not limit the number of open cursors. However, you cannot open a currently open cursor. Doing so results in an error message.

While an application executes, you can open a cursor as many times as necessary, but you must close it before reopening it. You need not retrieve all the rows from a cursor result set before retrieving rows from another cursor result set.

Fetching data using cursors

Use a fetch statement to retrieve data through a cursor and assign it to host variables. The syntax for the fetch statement is:

```
exec sql [at connect_name] fetch cursor_name
into : host_variable
[[ indicator]: indicator_variable ]
[, : host_variable
[[ indicator]: indicator_variable ]...];
```

where there is one *host_variable* for each column in the result rows.

Prefix each host variable with a colon and separate it from the next host variable with a comma. The host variables listed in the fetch statement must correspond to Adaptive Server Enterprise values that the select statement retrieves. Thus, the number of variables must match the number of returned values, they must be in the same order, and they must have compatible datatypes.

An *indicator_variable* is a 2-byte signed integer declared in a previous declare section. If a value retrieved from Adaptive Server Enterprise is null, the runtime system sets the corresponding indicator variable to -1. Otherwise, the indicator is set to 0.

The data that the fetch statement retrieves depends on the cursor position. The cursor points to the *current row*. The fetch statement always returns the current row. The first fetch retrieves the first row and copies the values into the host variables indicated. Each fetch advances the cursor to the next result row.

Normally, you should place the fetch statement within a loop so that all values returned by the select statement can be assigned to host variables.

The following loop uses the whenever not found statement:

```
/* Initialize error-handling routines */
exec sql whenever sqlerror call err_handle();
exec sql whenever not found goto end_label;
for (;;)
{
    exec sql fetch cursor_name
        into :host_variable [, host_variable];
    ...
}
end_label:
```

This loop continues until all rows are returned or an error occurs. In either case, `sqlcode` or `sqlstate`, which the whenever statement checks after each fetch, indicates the reason for exiting the loop. The error-handling routines ensure that an action is performed when either condition arises, as described in Chapter 8, “Handling Errors.”

Fetching data using scrollable cursors

Use a fetch statement to retrieve data through a cursor and assign it to host variables. The syntax for the fetch statement is:

```
exec sql [at connect_name] fetch [fetch
orientation] cursor_name
into : host_variable
[[ indicator]: indicator_variable ]
[, : host_variable
[[ indicator]: indicator_variable ]...];
```

where one *host_variable* exists for each column in the result rows.

Prefix each host variable with a colon, and separate it from the next host variable with a comma. The host variables listed in the `fetch` statement must correspond to Adaptive Server Enterprise values that the `select` statement retrieves. Thus, the number of variables must match the number of returned values, they must be in the same order, and they must have compatible datatypes.

The *fetch orientation* specifies the fetch direction of the row to be fetched, if a cursor is scrollable. The options are: `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE fetch_offset` and `RELATIVE fetch_offset`. If fetch orientation is not specified, `next` is default. If fetch orientation is specified, the cursor must be scrollable.

The data that the `fetch` statement retrieves depends on the cursor position. The `fetch` statement typically retrieves single or multiple rows from the cursor result set, depending on the `ROW_COUNT` specification at cursor open time. If a cursor is not scrollable, `fetch` retrieves the next row in the result set. If a cursor is scrollable, commands in the `fetch` statement specify the row position to be fetched.

Example for declaring a scrollable cursor and fetching rows

To declare a scrollable cursor and fetch rows at random, specify the scroll sensitivity and scrollability in the `declare cursor`, then specify the fetch orientation at fetch time. The following example demonstrates declaring an insensitive scrollable cursor and fetching rows at random:

```
exec sql declare c1 insensitive scroll cursor for
select title_id, royalty, ytd_sales from authors
where royalty < 25;
exec sql open c1;
```

In this example, `scroll` and `insensitive` are specified in the `declare cursor`. A fetch orientation can be specified at fetch time to indicate which row is required from the result set.

Once a cursor has been declared as scrollable and opened, a `FETCH` orientation can be specified at fetch time to indicate which row is wanted from the result set.

The following fetch example fetches the specified columns of the first row from the result set:

```
exec sql fetch first from c1 into :title,:roy,:sale;
```

The following fetch example fetches the specified columns of the previous row from the result set:

```
exec sql fetch prior from c1 into :title,:roy,:sale;
```

The following fetch example fetches the specified columns of row twenty from the result set:

```
exec sql fetch absolute 20 from c1 into :title, :roy, :sale;
```

Use *sqlcode* or *sqlstate* to determine if fetch statements return valid rows. For scrollable cursors, it is possible to fetch 0 rows if the cursor is positioned outside of result set boundaries, for example, before the first row or after the last row. In these circumstances, fetching 0 rows is expected.

Using cursors to update and delete rows

To update or delete the current row of a cursor, specify the where current of *cursor_name* as the search condition in an update or delete statement.

To update rows through a cursor, the result columns to be used in the updates must be updatable. They cannot be the result of SQL expressions such as `max(colname)`. In other words, there must be a valid correspondence between the result column and the database column to be updated.

The following example demonstrates how to use a cursor to update rows:

```
exec sql declare c1 cursor for
    select title_id, royalty, ytd_sales
    from titles
    where royalty < 25;

exec sql open c1;

for (;;)
{
    exec sql fetch c1 into :title, :roy, :sales;
    if (SQLCODE == 100) break;
    if (sales > 10000)
        exec sql update titles
            set royalty = :roy + 2
            where current of c1;
}
exec sql close c1;
```

The Embedded SQL syntax of the update and delete statements is the same as in Transact-SQL, with the addition of the where current of *cursor_name* search condition.

For details on determining table update protocol and locking, see the *Adaptive Server Enterprise Transact-SQL Users Guide*.

Closing cursors

Use the close statement to close an open cursor. The syntax for the close statement is:

```
exec sql [at connection] close cursor_name;
```

To reuse a closed cursor, issue another open statement. When you re-open a cursor, it points to the first row. Do not issue a close statement for a cursor that is not open or an error will result.

Cursor example

The following example shows how to nest two cursors. Cursor c2 depends upon the value fetched into *title-id* from cursor c1.

The program gets the value of *title-id* at open time, not at declare time.

```
exec sql include sqlca;
main()
{
    exec sql begin declare section;
        CS_CHAR title_id[7];
        CS_CHAR title[81];
        CS_INT  totalsales;
        CS_SMALLINT salesind;
        CS_CHAR au_lname[41];
        CS_CHAR au_fname[21];
    exec sql end declare section;
    exec sql whenever sqlerror call error_handler();
    exec sql whenever sqlwarning call error_handler();
    exec sql whenever not found continue;
    exec sql connect "sa" identified by "";
    exec sql declare c1 cursor for
        select title_id, title, total_sales from pubs2..titles;
    exec sql declare c2 cursor for
        select au_lname, au_fname from pubs2..authors
        where au_id in (select au_id from pubs2..titleauthor
            where title_id = :title_id);
    exec sql open c1;
    for (;;)
    {
        exec sql fetch c1 into :title_id, :title,
            :totalsales :salesind;
        if (sqlca.sqlcode == 100)
            break;
        printf("\nTitle ID: %s, Total Sales: %d", title_id, totalsales);
    }
}
```

```

    printf("\n%s", title);
if (totalsales > 10)
{
    exec sql open c2;
    for (;;)
    {
        exec sql fetch c2 into :au_lname, :au_fname;
        if (sqlca.sqlcode == 100)
            break;
        printf("\n\tauthor: %s, %s", au_lname, au_fname);
    }
    exec sql close c2;
}
}
exec sql close c1;
exec sql disconnect all;
}
error_handler()
{
printf("%d\n%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
exec sql disconnect all;
exit(0);
}

```

The following example is for insensitive scrollable cursors:

```

/*
**      example4.cp
**
**      This example is a non-interactive query program that
**      shows the user some actions executed by a scrollable,
**      insensitive cursor. This serves as a demo for usage
**      of scrollable cursors in ESQL/C.
*/
#include <stdio.h>
#include "sybsqllex.h"

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
#define TITLE_STRING      65
EXEC SQL END DECLARE SECTION;

void      error_handler();
void      warning_handler();
void      notfound_handler();

```

```
int
main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char    username[30];
    char    password[30];
    char    a_type[TITLE_STRING+1];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR CALL error_handler();
    EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
    EXEC SQL WHENEVER NOT FOUND CALL notfound_handler();

    strcpy(username, USER);
    strcpy(password, PASSWORD);

    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    EXEC SQL USE pubs2;

/*
** Declare an insensitive scrollable cursor against the
** titles table.
*/

    EXEC SQL DECLARE typelist INSENSITIVE SCROLL CURSOR FOR
    SELECT DISTINCT title FROM titles;

    EXEC SQL OPEN typelist;

    printf("\n==> Selecting the FIRST book Title:\n");

/*
** Fetch the first row in cursor resultset
*/
    EXEC SQL FETCH FIRST FROM typelist INTO :a_type;

    printf("\n%s\n", a_type);

/*
** Fetch the last row in cursor resultset
*/
    printf("\n==> Selecting the LAST book Title:\n");

    EXEC SQL FETCH LAST FROM typelist INTO :a_type;
```



```
printf("\n%s\n", a_type);

/*
** Fetch the previous (PRIOR) row based on current
** cursor position
*/
printf("\n==> Selecting the PREVIOUS book Title:\n");

EXEC SQL FETCH PRIOR FROM typelist INTO :a_type;

printf("\n%s\n", a_type);

/*
** Jump 5 rows back from current cursor position
*/
printf("\n==> Rewinding 5 STEPS through the Book
selection...\n");

EXEC SQL FETCH RELATIVE -5 FROM typelist INTO :a_type;

printf("\n%s\n", a_type);

/*
** Fetch the next row based on current cursor position
*/
printf("\n==> Selecting the NEXT book Title:\n");

EXEC SQL FETCH NEXT FROM typelist INTO :a_type;

printf("\n%s\n", a_type);

/*
** Jump out of the cursor result set. Note that this will
** lead to a "no rows found" condition. There are only 18
** rows in 'titles'.
*/

a_type[0] = '\0';

printf("\n==> Jumping out of the resultset.\n");

EXEC SQL FETCH ABSOLUTE 100 FROM typelist INTO :a_type;

printf("\n%s\n", a_type);

/* Close shop */
```

```
EXEC SQL CLOSE typelist;

printf("\n==> That's it for now.\n");

EXEC SQL DISCONNECT DEFAULT;

return(STDEXIT);
}

/* Error handlers deleted */
```

See the online sample programs for more examples using cursors. For details on accessing the online examples, see the *Open Client and Open Server Programmers Supplement* for your platform.

Using stored procedures

There are two types of *stored procedures*: user-defined and precompiler-generated. Both types run faster than standalone statements because Adaptive Server Enterprise preoptimizes the queries. You create user-defined stored procedures, and the precompiler generates stored procedures.

User-defined stored procedures

With Embedded SQL, you can execute stored procedures with `select` statements that return data rows. Stored procedures can return results to your program through output parameters and through a return status variable.

Stored procedure parameters can be either input, output, or both input and output. For details on stored procedures, see the *Adaptive Server Enterprise Transact-SQL Users Guide*.

Syntax

Valid stored procedure names consist of upper- and lowercase letters of the alphabet, “\$”, “_”, and “#”.

Do not include the `use` statement in a stored procedure.

To execute a stored procedure, use the following syntax:

```
exec [[:status_variable =] status_value] procedure_name
[[[@parameter_name=]parameter_value [out [put]]],...]
[into :hostvar_1 [:indicator_1]
[, hostvar_n [indicator_n, ...]]]
```

[with recompile];

where:

- *status_variable* can return either an Adaptive Server Enterprise return status value or a return code, which either indicates that the stored procedure completed successfully or gives the reasons for the failure. Negative status values are reserved for Adaptive Server Enterprise use. See the *Adaptive Server Enterprise Transact-SQL Users Guide* for a list of return status values for stored procedures.
- *status_value* is the value of the stored procedure return status variable *status_variable*.
- *procedure_name* is the name of the stored procedure to execute.
- *parameter_name* is the name of a variable in the stored procedure. You can pass parameters either by position or by name. If one parameter is named, all of them must be named. See the *Adaptive Server Enterprise Transact-SQL Users Guide*.
- *parameter_value* is a literal constant whose value is passed to the stored procedure.
- *output* indicates that the stored procedure returns a parameter value. The matching parameter in the stored procedure must also have been created using the *output* keyword.
- *into:hostvar_1* causes row data returned from the stored procedure to be stored in the specified host variables (*hostvar_1* through *hostvar_n*). Each host variable can have an indicator variable.
- *indicator_n* is a 2-byte host variable declared in a previous *declare* section. If the value for the associated *hostvar_n* is null, the indicator variable is set to -1 when the row data is retrieved. If truncation occurs, the indicator variable is set to the actual length of the result column. Otherwise, the indicator variable is 0.
- *with recompile* causes Adaptive Server Enterprise to create a new query plan for this stored procedure each time the procedure executes.

Note In Embedded SQL, the *exec* keyword is required to execute a stored procedure. You cannot substitute *execute* for *exec*.

Stored procedure example

The following example shows a call to a **stored procedure** where *retcode* is a status variable, *a_proc* is the stored procedure, *par1* is an input parameter, and *par2* is an output parameter:

```
exec sql begin declare section;
CS_INT  par1;
CS_INT  par2;
CS_SMALLINT  retcode;
exec sql end declare section;
...
exec sql exec :retcode = a_proc :par1, :par2 out;
```

The next example demonstrates the use of a stored procedure that retrieves data rows. The name of the stored procedure is *get_publishers*:

```
exec sql begin declare section;
CS_CHAR  pub_id(4);
CS_CHAR  name(45);
CS_CHAR  city(25);
CS_CHAR  state(2);
CS_SMALLINT  retcode;
exec sql end declare section;
. . .
exec sql exec :retcode = get_publishers :pub_id
                        into :name :city :state;
```

See Chapter 10, “Embedded SQL Statements: Reference Pages,” for a more detailed example of the `exec` statement.

Conventions

The datatypes of the stored procedure parameters must be compatible with the C host variables. Client-Library only converts certain combinations. See Chapter 4, “Using Variables,” for a table of compatible datatypes.

Precompiler-generated stored procedures

You can set an optional command line switch so that the precompiler automatically generates stored procedures that can optimize the execution of Transact-SQL statements in your program.

For the list of precompiler command line option switches, see the *Open Client and Open Server Programmers Supplement*.

Follow these steps to activate precompiler-generated stored procedures:

- 1 Set the appropriate command line switch so that the precompiler automatically generates stored procedures for the Transact-SQL statements to be optimized.

The precompiler generates an isql file containing statements that generate the stored procedures.

- 2 Use interactive SQL (the isql program) to execute the file.

This loads the stored procedures on Adaptive Server Enterprise. The precompiler also creates the stored procedure calls in its output file.

By default, precompiler-generated stored procedures have the same name as the source program, minus any file extensions. The stored procedures are numbered sequentially and the file name and number are separated by a semicolon (“;”).

For example, the stored procedures for a source program named *test1.pc*, would be named *test1;1* through *test1;n*, where *n* is the number of the source program’s last stored procedure.

Optionally, you can set a command line flag that lets you alter the stored procedures’ names. By using this flag, you can test a modified application without deleting a stored procedure already in production. After successfully testing the application, you can precompile it without the flag to install the stored procedure.

Note When you issue the declare cursor statement, only the select clause is saved as a stored procedure. If an application has syntax errors, the precompiler generates neither the target file nor stored procedures.

Grouping statements

Statements can be grouped for execution by batch or by transactions.

Grouping statements by batches

A batch is a group of statements you submit as one unit for execution. The precompiler executes all Transact-SQL statements between the `exec sql` and `;` keywords in batch mode.

Although the precompiler saves stored procedures, it does not save batches for re-execution. The batch is effective only for the current execution.

The precompiler supports only batch mode statements that return no result sets.

```
exec sql insert into TABLE1 values (:val1)
       insert into TABLE2 values (:val2)
       insert into TABLE3 values (:val3);
```

The three insert statements are processed as a group, which is more efficient than being processed individually. Use the `get diagnostics` method of error handling with batches. For details, see “Using `get diagnostics`” on page 97.

These statements are legal within a batch because none of them returns results. See the *Adaptive Server Enterprise Transact-SQL Users Guide*.

Grouping statements by transactions

A **transaction** is a single unit of work, whether the unit consists of one or 100 statements. The statements in the transaction execute as a group, so either all or none of them execute.

The precompiler supports two transaction modes: default ANSI/ISO and optional Transact-SQL. In the Transact-SQL transaction mode, each statement is implicitly committed unless it is preceded by a `begin` transaction statement.

The Transact-SQL mode uses relatively few system resources, while the default ANSI/ISO transaction mode can dramatically affect system response time. For details on choosing the appropriate mode for your application, see the *Adaptive Server Enterprise Transact-SQL Users Guide*.

You can use a precompiler option to determine the transaction mode of the connections your application opens. See the *Open Client and Open Server Programmers Supplement* for details.

Transact-SQL transaction mode

In this optional Transaction mode, the Embedded SQL syntax is the same as that used in Transact-SQL. The `begin` transaction statement explicitly initiates transactions.

The syntax of the Embedded SQL transaction statements is:

```
exec sql [at connect_name ]
       begin transaction [ transaction_name ];
```

```
exec sql [at connect_name]
    save transaction [ savepoint_name];

exec sql [at connect_name] commit transaction
    [ transaction_name ];

exec sql [at connect_name] rollback transaction
    [ savepoint_name | transaction_name ];
```

Note The disconnect command rolls back all open transactions. For details on this statement, see Chapter 5, “Connecting to Adaptive Server Enterprise.”

When you issue a begin transaction on a connection, you must also issue a save, commit, or roll back transaction on the same connection. Otherwise, an error is generated.

Default ANSI/ISO transaction mode

ANSI/ISO SQL does not provide a save transaction or begin transaction statement. Instead, transactions begin implicitly when the application program executes one of the following statements:

- delete
- insert
- select
- update
- open
- exec

The transaction ends explicitly when you issue either a commit work or rollback work statement. You must use the ANSI/ISO forms of the commit and rollback statements. The syntax is:

```
exec sql commit [work] end-exec
exec sql rollback [work] end-exec
```

Extended transactions

An **extended transaction** is a unit of work that has multiple Embedded SQL statements. In the Transact-SQL **transaction mode**, you surround an extended transaction statement with the begin transaction and commit transaction statements.

In the default ANSI mode, you are constantly within an extended transaction. When you issue a commit work statement, the current extended transaction ends and another begins. For details, see the *Adaptive Server Enterprise Transact-SQL Users Guide*.

Note Unless the database option `allow ddl in tran` is set, do not use the following Transact-SQL statements in an extended, ANSI-mode transaction: `alter database`, `create database`, `create index`, `create table`, `create view`, `disk init`, `grant`, `load database`, `load transaction`, `revoke`, `truncate table`, and `update statistics`.

Using Dynamic SQL

This chapter explains dynamic SQL, an advanced methodology that lets users of Embedded SQL applications enter SQL statements while the application is running. While static SQL will suffice for most of your needs, dynamic SQL provides the flexibility to build diverse SQL statements at runtime.

Topic	Page
Dynamic SQL overview	74
Dynamic SQL protocol	75
Method 1: Using execute immediate	76
Method 2: Using prepare and execute	77
Method 3: Using prepare and fetch with a cursor	80
Method 4: Using prepare and fetch with dynamic descriptors	84
Summary	92

Dynamic SQL is a set of Embedded SQL statements that permit users of online applications to access the database interactively at application runtime.

Use dynamic SQL when one or more of the following conditions is not known until runtime:

- SQL statement the user will execute
- Column, index, and table references
- Number of host variables, or their datatypes

Dynamic SQL overview

Dynamic SQL is part of ANSI and the ISO SQL2 standard. It is useful for running an interactive application. If the application only accepts a small set of SQL statements, you can embed them within the program. However, if the application accepts many types of SQL statements, you can benefit from constructing SQL statements, then binding and executing them dynamically.

The following situation would benefit from use of dynamic SQL: The application program searches a bookseller's database of books for sale. A potential buyer can apply many criteria, including price, subject matter, type of binding, number of pages, publication date, language, and so on.

A customer might say, "I want a nonfiction book about business that costs between \$10 and \$20." This request is readily expressed as a Transact-SQL statement:

```
select * from titles where
    type = "business"
    and price between $10 and $20
```

It is not possible to anticipate the combinations of criteria that all buyers will apply to their book searches. Therefore, without using dynamic SQL, an Embedded SQL program could not easily generate a list of prospective books with a single query.

With dynamic SQL, the bookseller can enter a **query** with a different where clause search condition for each buyer. The seller can vary requests based on the publication date, book category, and other data, and can vary the columns to be displayed. For example:

```
select * from titles
    where type = ?
    and price between ? and ?
```

The question marks ("?") are dynamic parameter markers that represent places where the user can enter search values.

Dynamic SQL protocol

Note The precompiler does not generate stored procedures for dynamic SQL statements because the statements are not complete until runtime. At runtime, Adaptive Server Enterprise stores them as temporary stored procedures in the tempdb database. The tempdb database must contain the user name "guest", which in turn must have create procedure permission. Otherwise, attempting to execute one of these temporary stored procedures generates the error message, "Server user id *user_id* is not a valid user in database *database_name*," where *user_id* is the user's user ID, and *database_name* is the name of the user's database.

The dynamic SQL prepare statement sends the actual SQL statement, which can be any Data Definition Language (DDL) or Data Manipulation Language (DML) statements, or any Transact-SQL statement except create procedure.

The dynamic SQL facility performs these actions:

- 1 Translates the input data into a SQL statement.
- 2 Verifies that the SQL statement can execute dynamically.
- 3 Prepares the SQL statement for execution, sending it to Adaptive Server Enterprise, which compiles and saves it as a temporary stored procedure (for methods 2, 3, and 4).
- 4 Binds all input parameters or descriptor (for methods 2, 3, and 4).
- 5 Executes the statement.
For a varying-list select, it uses a descriptor to reference the data items and rows returned (for method 4).
- 6 Binds the output parameters or descriptor (for methods 2, 3, and 4).
- 7 Obtains results.
- 8 Drops the statement (for methods 2, 3, and 4) by deactivating the stored procedure in Adaptive Server Enterprise.
- 9 Handles all error and warning conditions from Adaptive Server Enterprise and Client-Library.

Method 1: Using *execute immediate*

Use *execute immediate* to send a complete Transact-SQL statement, stored in a host variable or literal string, to Adaptive Server Enterprise. The statement cannot return any results—you cannot use this method to execute a *select* statement.

The dynamically entered statement executes as many times as the user invokes it during a session. With this method:

- 1 The Embedded SQL program passes the text to Adaptive Server Enterprise.
- 2 Adaptive Server Enterprise verifies that the statement is not a *select* statement.
- 3 Adaptive Server Enterprise compiles and executes the statement.

With *execute immediate*, you can let the user enter all or part of a Transact-SQL statement.

The syntax for *execute immediate* is:

```
exec sql [at connection_name] execute immediate  
  {:host_variable | string};
```

where:

- *host_variable* is a character-string variable defined in a *declare* section. Before calling *execute immediate*, the host variable should contain a complete and syntactically correct Transact-SQL statement.
- *string* is a literal Transact-SQL statement string that can be used in place of *host_variable*.

Embedded SQL sends the statement in *host_variable* or string to Adaptive Server Enterprise without any processing or checking. If the statement attempts to return results or fails, an error occurs. You can test the value of `SQLCODE` after executing the statement or use the *whenever* statement to set up an error handler. See Chapter 8 for information about handling errors in Embedded SQL programs.

Method 1 examples

The following two examples demonstrate using method 1, *execute immediate*. The first example prompts the user to enter a statement and then executes it:

```

exec sql begin declare section;
  CS_CHAR statement_buffer[linesize];
exec sql end declare section;
...
printf("\nEnter statement\n");
gets(statement_buffer);

exec sql [at connection] execute immediate
:statement_buffer;

```

The next example prompts the user to enter a search condition to specify rows in the titles table to update. Then, it concatenates the search condition to an update statement and sends the complete statement to Adaptive Server Enterprise.

```

exec sql begin declare section;
  CS_CHAR sqlstring[200];
exec sql end declare section;

char      cond[150];

exec sql whenever sqlerror call err_p();
exec sql whenever sqlwarning call warn_p();

strcpy(sqlstring,
"update titles set price=price*1.10 where ");

printf("Enter search condition:");
scanf("%s", cond);
strcat(sqlstring, cond);

exec sql execute immediate :sqlstring;

exec sql commit work;

```

Method 2: Using *prepare* and *execute*

Use method 2, prepare and execute, when one of the following cases is true:

- You are certain that no data will be retrieved, and you want the statement to execute more than once.
- A select statement is to return a single row. With this method, you cannot associate a cursor with the select statement.

This process is also called a single-row select. If a user needs to retrieve multiple rows, use method 3 or 4.

This method uses prepare and execute to substitute data from C variables into a Transact-SQL statement before sending the statement to Adaptive Server Enterprise. The Transact-SQL statement is stored in a character buffer with dynamic parameter markers to show where to substitute values from C variables.

Because this statement is prepared, Adaptive Server Enterprise compiles and saves it as a temporary stored procedure. Then, the statement executes repeatedly, as needed, during the session.

The prepare statement associates the buffer with a statement name and prepares the statement for execution. The execute statement substitutes values from a list of C variables or SQL descriptors into the buffer and sends the completed statement to Adaptive Server Enterprise. You can execute any Transact-SQL statement this way.

prepare

The syntax for the prepare statement is:

```
exec sql [at connection] prepare statement_name from  
{:host_variable | string};
```

where:

- *statement_name* is a name up to 255 characters long that identifies the statement. It is a symbolic name or a C character array host variable containing the name of the statements that the precompiler uses to associate an execute statement with a prepare statement.

- *host_variable* is a character array host variable.

Precede the host variable with a colon, as in standard Embedded SQL statements.

- *string* is a literal string that can be used in place of *host_variable*.

host_variable or *string* can contain dynamic parameter markers (“?”), which indicate places in the dynamic query where values will be substituted when the statement executes.

execute

The syntax for the execute statement is:

```

exec sql [at connection] execute statement_name
    [into host_var_list | sql descriptor
        descriptor_name | descriptor sqllda_name]
    [using host_var_list | sql descriptor
        descriptor_name | descriptor sqllda_name];

```

where:

- *statement_name* is the name assigned in the prepare statement.
- *into* is used for a single-row select.
- *using* specifies the C variables or descriptors substituted for a dynamic parameter marker in *host_variable*. The variables, which you must define in a declare section, are substituted in the order listed. You need only this clause when the statement contains dynamic parameter markers.
- *descriptor_name* represents the area of memory that holds a description of the dynamic SQL statement's dynamic parameter markers.
- *host_var_list* a list of host variables to substitute into the parameter markers (“?”) in the query.
- *sqllda_name* is the name of the SQLDA.

Method 2 example

The following example demonstrates using prepare and execute in method 2. This example prompts the user to enter a where clause that determines which rows in the titles table to update and a multiplier to modify the price. According to what the user elects, the appropriate string is concatenated to the update statement stored in host variable “*sqlstring*”.

```

exec sql begin declare section;

        CS_CHAR sqlstring[200];
        CS_FLOAT multiplier;

exec sql end declare section;

```

```
char    cond[150];

exec sql whenever sqlerror perform err_p();
exec sql whenever sqlwarning perform warn_p();
printf("Enter search condition:");
scanf("%s", cond);
printf("Enter price multiplier: ");
scanf("%f", &multiplier);
strcpy(sqlstring,
       "update titles set price = price * ? where ");
strcat(sqlstring, cond);
exec sql prepare update_statement from :sqlstring;
exec sql execute update_statement using
       :multiplier;
exec sql commit;
```

Method 3: Using *prepare* and *fetch* with a cursor

Method 3 uses the prepare statement with cursor statements to return results from a select statement. Use this method for fixed-list select statements that may return multiple rows. That is, use it when the application has determined in advance the number and type of select column list attributes to be returned. You must anticipate and define host variables to accommodate the results.

When you use method 3, include the declare, open, fetch, and close cursor statements to execute the statement. This method is required because the statement returns more than one row. There is an association between the prepared statement identifier and the specified cursor name. You can also include update and delete where current of cursor statements.

As with method 2, prepare and execute, a Transact-SQL select statement is first stored in a character host variable or string. It can contain dynamic parameter markers to show where to substitute values from input variables. The statement is given a name to identify it in the prepare, declare, and open statements.

Method 3 requires five steps:

- 1 prepare
- 2 declare
- 3 open
- 4 fetch (and, optionally, update and delete)

5 close

These steps are described in the following sections.

prepare

The prepare statement is the same as that used with method 2. For details, see “prepare” on page 78.

declare

The declare statement is similar to the standard declare statement for cursors. In dynamic SQL, however, you declare the cursor for a prepared *statement_name* instead of for a select statement, and any input host variables are referenced in the open statement instead of in the declare statement.

A dynamic declare statement is an executable statement rather than a declaration. As such, it must be positioned in the code where executable statements are legal, and the application should check status codes (SQLCODE, SQLCA, or SQLSTATE) after executing the declaration.

The dynamic SQL syntax for the declare statement is:

```
exec sql [at connection_name] declare cursor_name
        cursor for statement_name;
```

where:

- *at connection_name* specifies the Adaptive Server Enterprise connection the cursor will use.
- *cursor_name* identifies the cursor, used with the open, fetch, and close statements.
- *statement_name* is the name specified in the prepare statement, and represents the select statement to be executed.

open

The open statement substitutes any input variables in the statement buffer, and sends the result to Adaptive Server Enterprise for execution. The syntax for the open statement is:

```
exec sql [at connection_name] open cursor_name [using  
{host_var_list | sql descriptor descriptor_name |  
descriptor sqllda_name}];
```

where:

- *cursor_name* is the name given to the cursor in the declare statement.
- *host_var_list* consists of the names of the host variables that contain the values for dynamic parameter markers.
- *descriptor_name* is the name of the descriptor that contains the value for the dynamic parameter markers.
- *sqllda_name* is the name of the SQLDA.

fetch and close

After a cursor opens, the result sets are returned to the application. Then, the data is fetched and loaded into the application program host variables. Optionally, you can update or delete the data. The fetch and close statements are the same as in static Embedded SQL.

The syntax for the fetch statement is:

```
exec sql [at connection_name] fetch cursor_name into  
:host_variable [[indicator]:indicator_variable]  
[, :host_variable  
[[indicator]:indicator_variable]...];
```

where:

- *cursor_name* is the name given to the cursor in the declare statement.
- There is one *C host_variable* for each column in the result rows. The variables must have been defined in a declare section, and their datatypes must be compatible with the results returned by the cursor.

The syntax for the close statement is:

```
exec sql [at connection_name] close cursor_name;
```

where *cursor_name* is the name assigned to the cursor in the declare statement.

Method 3 example

The following example uses prepare and fetch, and prompts the user for an order by clause in a select statement:

```

exec sql begin declare section;
  CS_CHAR      sqlstring[200];
  CS_FLOAT     bookprice,condprice;
  CS_CHAR      booktitle[200];
exec sql end declare section;

char          orderby[150];

exec sql whenever sqlerror call err_p();
exec sql whenever sqlwarning call warn_p();

strcpy(sqlstring,
"select title,price from titles\
where price>? order by ");

printf("Enter the order by clause:");
scanf("%s", orderby);
strcat(sqlstring, orderby);

exec sql prepare select_state from :sqlstring;
exec sql declare select_cur cursor for select_state;

condprice = 10; /* the user can be prompted
                ** for this value */

exec sql open select_cur using :condprice;
exec sql whenever not found goto end;

for (;;)
{
  exec sql fetch select_cur
    into :booktitle,:bookprice;
  printf("%20s  %bookprice=%6.2f\n",
    booktitle, bookprice);
}

end:

exec sql close select_cur;
exec sql commit work;

```

Method 4: Using *prepare* and *fetch* with dynamic descriptors

Method 4 permits varying-list select statements. That is, when you write the application, you need not know the formats and number of items the select statement will return. Use method 4 when you cannot define the host variables in advance because you do not know how many variables are needed or of what type they should be.

Method 4 dynamic descriptors

A **dynamic descriptor** is a data structure that holds a description of the variables used in a dynamic SQL statement. There are two kinds of dynamic descriptors—SQL descriptors and SQLDA structures. Both are described later in this chapter.

When a cursor opens, it can have an input descriptor associated with it. The input descriptor contains the values to be substituted for the dynamic SQL statement's parameter markers.

Before the cursor is opened, the user fills in the input descriptor with the appropriate information, including the number of parameters, and, for each parameter, its type, length, precision, scale, indicator, and data.

Associated with the fetch statement is an output descriptor, which holds the resultant data. Adaptive Server Enterprise fills in the data item's attributes, including its type and the actual data being returned. If you are using an SQL descriptor, use the get descriptor statement to copy the data into host variables.

Dynamic SQL method 4 performs the following steps:

- 1 Prepares the statement for execution.
- 2 Associates a cursor with the statement.
- 3 Defines and binds the input parameters or descriptor and:
 - If using an input descriptor, allocates it
 - If using an input parameter, associates it with the statement or cursor
- 4 Opens the cursor with the appropriate input parameter(s) or descriptor(s).
- 5 Allocates the output descriptor if different from the input descriptor and binds the output descriptor to the statement.
- 6 Retrieves the data by using fetch cursor and the output descriptor.

- 7 Copies data from the dynamic descriptor into host program variables. If you are using an SQLDA, this step does not apply; the data is copied in step 6.
- 8 Closes the cursor.
- 9 Deallocates the dynamic descriptor(s).
- 10 Drops the statement (ultimately, the stored procedure).

Dynamic descriptor statements

There are statements that associate the descriptor with a SQL statement and with a cursor associated with the SQL statement. The following list describes dynamic SQL statements for method 4:

Statement	Description
allocate descriptor	Notifies Client-Library to allocate a SQL descriptor.
describe input	Obtains information about the dynamic parameter marker in the prepare statement.
set descriptor	Inserts or updates data in the system descriptor.
get descriptor	Moves row or parameter information stored in a descriptor into host variables, thereby allowing the application program to use the information.
execute	Executes a prepared statement.
open cursor	Associates a descriptor with a cursor and opens the cursor.
describe output	Obtains information about the select list columns in the prepared dynamic SQL statement.
fetch cursor	Retrieves a row of data for a dynamically declared cursor.
deallocate descriptor	Deallocates a dynamic descriptor.

For complete descriptions of these statements, see Chapter 10, “Embedded SQL Statements: Reference Pages.”

About SQL descriptors

A SQL descriptor is an area of memory that stores a description of the variables used in a prepared dynamic SQL statement. A SQL descriptor can contain the following information about data attributes (for details, see the descriptions of the set descriptor and get descriptor commands in Chapter 10, “Embedded SQL Statements: Reference Pages”):

- precision – integer.
- scale – integer.
- nullable – 1 (`cs_true`) if the column can contain nulls; 0 (`cs_false`) if it cannot. Valid only with get descriptor statement.
- indicator – value for the indicator associated with the dynamic parameter marker. Valid only with get descriptor statement.
- name – name of the dynamic parameter marker. Valid only with get descriptor statement.
- data – value for the dynamic parameter marker specified by the item number. If the value of *indicator* is -1, the value of *data* is undefined.
- count – number of dynamic parameter markers described in the descriptor.
- type – datatype of the dynamic parameter marker or host variable.
- returned_length – actual length of the data in an output column.

Method 4 example using SQL descriptors

The following example uses prepare and fetch with dynamic parameter markers and SQL descriptors.

```
exec sql begin declare section
    int  index_colcnt, coltype;
    int  int_buff;
    char char_buff[255], void_buff[255];
    char type[255], title[255];
    char colname[255];
    int  sales;
    int  descnt, occur, cnt;
    int  condcnt, diag_cnt, num_msgs;
    char user_id[30], pass_id[30], server_name[30];
    char str1[1024], str2[1024], str3[1024],
        str4[1024];
exec sql end declare section;
```

```
...
void dyn_m4()
{
    printf("\n\nDynamic sql Method 4\n");
    printf("Enter in a Select statement to retrieve
        any kind of ");
    printf("information from the pubs database:");
    scanf("%s", &str4);

    printf("\n\nEnter the largest number of columns to
        be retrieved or the number ");
    printf("of ? in the sql statement:\n");
    scanf("%d", &occur);

    exec sql allocate descriptor dinout with max
        :occur;
    exec sql prepare s4 from :str4;
    exec sql declare c2 cursor for s4;

    exec sql describe input s4 using sql descriptor
        dinout;

    fill_descriptor();

    exec sql open c2 using sql descriptor dinout;

    while (sqlca.sqlcode == 0)
    {
        exec sql fetch c2 into sql descriptor dinout;
        if(sqlca.sqlcode == 0) {
            print_descriptor();
        }
    }

    exec sql close c2;

    exec sql deallocate descriptor dinout;
    exec sql deallocate prepare s4;

    printf("Dynamic SQL Method 4 completed\n\n");

}

void
print_descriptor()
{
    exec sql get descriptor dinout :descnt = count;
```

```
printf("Column name \t\tColumn data\n");
printf("----- \t\t-----\n");

for (index_colcnt = 1; index_colcnt <= descnt;
     index_colcnt++)
{ /* get each column attribute */
  exec sql get descriptor dinout value
           :index_colcnt :coltype = TYPE;

  switch(coltype)
  {
    ...
    case 4:/* integer type */
      exec sql get descriptor dinout value
               :index_colcnt
               :colname = NAME, :int_buff = DATA;
      printf("%s \t\t %d\n", colname, int_buff);
      break;
    ...
  }
}

void
fill_descriptor()
{
  exec sql get descriptor dinout :descnt = count;
  for (cnt = 1; cnt <= descnt; cnt++)
  {
    printf("Enter in the data type of the %d ?:",
           cnt);
    scanf("%d", &coltype);
    switch(coltype)
    {
      ...
      case 4:/* integer type */
        printf("Enter in the value of the data:");
        scanf("%d\n", &int_buff);
        exec sql set descriptor dinout VALUE :cnt
                  TYPE = :coltype,
                  DATA = :int_buff;
        break;

      default:
```



```

        printf("non-supported column type.\n");
        break;
    }
}
}

```

About SQLDAs

SQLDA is a host-language structure that, like an SQL descriptor, describes the variables used in a dynamic SQL prepared statement. Unlike SQL descriptors, SQLDAs are public data structures whose fields you can access. Statements using SQLDAs may execute faster than equivalent statements using SQL descriptors.

The SQLDA structure is not part of the SQL standard. Different implementations of Embedded SQL define the SQLDA structure differently. Embedded SQL version 11.1 and later supports the SQLDA defined by Sybase; it does not support SQLDA datatypes defined by other vendors.

To define the SQLDA datatype in your Embedded SQL program, you use the Embedded SQL command `include sqlda`. To allocate a SQLDA structure in your program, you use the `malloc` function. To deallocate an SQLDA, you use the `free` function. Your program is responsible for deallocating all SQLDA structures that it creates. Embedded SQL does not limit the number of SQLDA structures that can be created by a program.

Table 7-1 describes the fields of the SQLDA structure.

Table 7-1: Fields of the SQLDA structure

Field	Datatype	Description
<code>sd_sqln</code>	CS_SMALLINT	The size of the <code>sd_column</code> array.
<code>sd_sqld</code>	CS_SMALLINT	The number of columns in the query being described, or 0 if the statement being described is not a query. For <code>fetch</code> , <code>open</code> , and <code>execute</code> statements, this field indicates the number of host variables described by occurrences of <code>sd_column</code> , or the number of dynamic parameter markers for the describe input statement.
<code>sd_column[].sd_datafmt</code>	CS_DATAFMT	Identifies the Client-Library CS_DATAFMT structure associated with this column. Refer to descriptions of <code>ct_bind</code> , <code>ct_param</code> , and <code>ct_describe</code> in the <i>Open Client Client-Library/C Reference Manual</i> .
<code>sd_column[].sd_sqldata</code>	CS_VOID	For <code>fetch</code> , <code>open</code> , and <code>execute</code> statements, stores the address of the statement's host variable. This field is not used for <code>describe</code> or <code>prepare</code> statements.

Field	Datatype	Description
sd_column[].sd_sqlind	CS_SMALLINT	For fetch, open, and execute statements, this field acts as an indicator variable for the column being described. If the column's value is null, this field is set to -1. This field is not used for describe or prepare statements.
sd_column[].sd_sqllen	CS_INT	The actual size of the data pointed to by <i>sd_sqldata</i> associated with this column.
sd_column[].sd_sqlmore	CS_VOID	Reserved.

The Embedded SQL header file *sqlda.h* contains a macro, `SQLDADECL`, that lets you declare `SQLDA` structures in your program. The `SQLDADECL` macro is as follows:

```
#ifndef SQLDADECL
#define SQLDADECL(name, size)
    struct {
        CS_INT      sd_sqln;
        CS_INT      sd_sqln;
        struct {
            CS_DATAFMT      sd_datafmt;
            CS_VOID          sd_sqldata;
            CS_SMALLINT     sd_sqlind;
            CS_INT           sd_sqllen;
            CS_VOID          sd_sqlmore;
        } sd_column[ (SIZE) ]
    } name
#endif /* SQLDADECL */
```

Method 4 example using SQLDAs

Following is an example that uses prepare and fetch with dynamic parameter markers and SQL descriptors.

```
exec sql include sqlca;
exec sql include sqlda;

...
SQLDA *input_descriptor, *output_descriptor;
CS_SMALLINT small;
CS_CHAR      character[20];

input_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
input_descriptor->sqlda_sqln = 3;
output_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
output_descriptor->sqlda_sqln = 3;
```

```

*p_retcode = CS_SUCCEED;
exec sql connect "sa" identified by "";
/* setup */
exec sql drop table example;
exec sql create table example (fruit char(30), number int);
exec sql insert example values ('tangerine', 1);
exec sql insert example values ('pomegranate', 2);
exec sql insert example values ('banana', 3);
/* Prepare and describe the select statement */
    exec sql prepare statement from
        "select fruit from example where number = ?";
exec sql describe input statement using descriptor    input_descriptor;
input_descriptor->sqlda_column[0].sqlda_datafmt.datatype =
CS_SMALLINT TYPE;
input_descriptor->sqlda_column[0].sqlda_sqldata = &small;
input_descriptor->sqlda_column[0].sqlda_sqlllen = sizeof(small);
small = 2;
exec sql describe output statement using descriptor
    output_descriptor;
if (output_descriptor->sqlda_sqld != 1 ||
    output_descriptor->sqlda_column[0].sqlda_datafmt.datatype !=
CS_CHAR_TYPE)
    FAIL;
else
    printf("First describe output \n");
output_descriptor->sqlda_column[0].sqlda_sqldata = character;
output_descriptor->sqlda_column[0].sqlda_datafmt.maxlength = 20;
exec sql execute statement into descriptor output_descriptor
    using descriptor input_descriptor;
printf("Expected pomegranate, got %s\n", character);
exec sql deallocate prepare statement;
/* Prepare and describe second select statement */
exec sql prepare statement from
    "select number from example where fruit = ?";
exec sql declare c cursor for statement;
exec sql describe input statement using descriptor
    input_descriptor;
input_descriptor->sqlda_column->sqlda_sqldata = character;
input_descriptor->sqlda_column->sqlda_datafmt.maxlength =      CS_NULLTERM;
strcpy(character, "banana");
input_descriptor->sqlda_column->sqlda_sqlllen = CS_NULLTERM;
exec sql open c using descriptor input_descriptor;
exec sql describe output statement using descriptor
    output_descriptor;
output_descriptor->sqlda_column->sqlda_sqldata = character;
output_descriptor->sqlda_column->sqlda_datafmt.datatype =      CS_CHAR_TYPE;

```

```
output_descriptor->sqlda_column->sqlda_datafmt.maxlength = 20;
output_descriptor->sqlda_column->sqlda_sqlllen = 20;
output_descriptor->sqlda_column->sqlda_datafmt.format =
    (CS_FMT_NULLTERM | CS_FMT_PADBLANK);
exec sql fetch c into descriptor output_descriptor;
printf("Expected pomegranate, got %s\n", character);
exec sql commit work;
```

Summary

This chapter described dynamic SQL, a set of Embedded SQL statements that permit online applications to access the database interactively. This interaction with the database lets a user define and execute SQL statements at runtime.

The four dynamic SQL Methods are:

- Method 1: execute immediate
- Method 2: prepare and execute
- Method 3: prepare and fetch
- Method 4: prepare and fetch with dynamic descriptors

The next chapter describes how to detect and correct Embedded SQL errors.

Handling Errors

This chapter discusses how to detect and correct errors that can occur during the execution of Embedded SQL programs. It covers the `whenever` and `get diagnostics` statements, which you can use to process warnings and errors, and the `SQLCA` variables that pertain to warnings and errors.

Topic	Page
Testing for errors	94
Testing for warning conditions	94
Trapping errors with <code>whenever</code>	95
Using <code>get diagnostics</code>	97
Writing routines to handle warnings and errors	98
Precompiler-detected errors	99

While an Embedded SQL application is running, some events may occur that interfere with the application's operation. Following are examples:

- Adaptive Server Enterprise becomes inaccessible.
- The user enters an incorrect password.
- The user does not have access to a database object.
- A database object is deleted.
- A column's datatype changes.
- A query returns an unexpected null value.
- A dynamic SQL statement contains a syntax error.

You can anticipate these events by writing warning and error-handling code to recover gracefully when one of these situations occurs.

Testing for errors

Embedded SQL places a return code in the *SQLCODE* variable to indicate the success or failure of each SQL statement sent to Adaptive Server Enterprise. You can either test the value of *SQLCODE* after each Embedded SQL statement or use the *whenever* statement to instruct the precompiler to write the test code for you. The *whenever* statement is described later in this chapter.

Using SQLCODE

Table 8-1 lists the values *SQLCODE* can contain:

Table 8-1: SQLCODE return values

Value	Meaning
0	No warnings or errors occurred.
<0	An error occurred. The <i>SQLCA</i> variables contain useful information for diagnosing the error.
100	No rows returned from last statement, although the statement executed successfully. This condition is useful for driving a loop that fetches rows from a cursor. When <i>SQLCODE</i> becomes 100, the loop and all rows that have been fetched end. This technique is illustrated in Chapter 6, "Using Transact-SQL Statements."

Testing for warning conditions

Even when *SQLCODE* indicates that a statement has executed successfully, a warning condition may still have occurred. The 8-character array *sqlca.sqlwarn* indicates such warning conditions. Each *sqlwarn* array element, or flag, stores either the space character or the character "W."

Table 8-2 describes what the space character or "W" means in each flag:

Table 8-2: sqlwarn flags

Flag	Description
sqlwarn[0]	If blank, no warning condition of any kind occurred, and all other sqlwarn flags are blank. If sqlwarn[0] is set to “W,” one or more warning conditions occurred, and at least one other flag is set to “W.”
sqlwarn[1]	If set to “W,” the character string variable that you designated in a fetch statement was too short to store the statement’s result data, so the result data was truncated. You designated no indicator variable to receive the original length of the data that was truncated.
sqlwarn[2]	If set to “W,” the input sent to Adaptive Server Enterprise contained a null value in an illegal context, such as in an expression or as an input value to a table that prohibits null values.
sqlwarn[3]	The number of columns in a select statement’s result set exceeds the number of host variables in the statement’s into clause.
sqlwarn[4]	Reserved.
sqlwarn[5]	Adaptive Server generated a conversion error while attempting to execute this statement.
sqlwarn[6]	Reserved.
sqlwarn[7]	Reserved.

Test for a warning after you determine that a SQL statement executed successfully. Use the whenever statement, as described in the next section, to instruct the precompiler to write the test code for you.

Trapping errors with *whenever*

Use the Embedded SQL whenever statement to trap errors and warning conditions. It specifies actions to be taken depending on the outcome of each Embedded SQL statement sent to Adaptive Server Enterprise.

The whenever statement is not executable. Instead, it directs the precompiler to generate C code that tests for specified conditions after each executable Embedded SQL statement in the program.

The syntax of the whenever statement is:

```
exec sql whenever {sqlwarning | sqlerror | not found}
    {continue | goto label |
```

```
call function_name ([param [, param]...]) | stop};
```

whenever testing conditions

Each whenever statement can test for one of the following three conditions:

- sqlwarning
- sqlerror
- not found

The precompiler generates warning messages if you do not write a whenever statement for each of the three conditions. If you write your own code to check for errors and warnings, suppress the precompiler warnings by writing a whenever...continue clause for each condition. This instructs the precompiler to ignore errors and warnings.

If you precompile with the verbose option, the precompiler generates a ct_debug() function call as part of each connect statement. This causes Client-Library to display informational, warning, and error messages to your screen as your application runs. The whenever statement does not disable these messages. See the *Open Client and Open Server Programmers Supplement*.

After an Embedded SQL statement executes, the values of *sqlcode* and *sqlwarn0* determine if one of the conditions exists. Table 8-3 shows the criteria whenever uses to detect the conditions:

Table 8-3: Criteria for the whenever statement

Condition	Criteria
sqlwarning	sqlcode = 0 and sqlwarn[0] = W
sqlerror	sqlcode < 0
not found	sqlcode = 100

To change the action of a whenever statement, write a new whenever statement for the same condition. whenever applies to all Embedded SQL statements that follow it, up to the next whenever statement for the same condition.

The whenever statement ignores the application program's logic. For example, if you place whenever at the end of a loop, it does not affect the preceding statements in subsequent passes through the loop.

whenever actions

The whenever statement specifies one of the following four actions:

Table 8-4: whenever actions

Action	Description
continue	Perform no special action when a SQL statement returns the specified condition. Normal processing continues.
goto	Perform a branch to an error handling procedure within your application program. You can write goto as either goto or goto, and you must follow it with a valid statement label name. The precompiler does not detect an error if the label name is not defined in the program, but the C compiler does.
call	Call another C routine and, optionally, pass variables.
stop	Terminate the program when a SQL statement triggers the specified condition.

Using get diagnostics

The get diagnostics statement retrieves error, warning, and informational messages from Client-Library. It is similar to, but more powerful than, the whenever statement, because you can expand it to retrieve more details of the detected errors.

If, within a whenever statement, you specify the application to go to or call another routine, specify get diagnostics in the function code, as follows:

```
void
error_handler()
{
    exec sql begin declare section;
        int num_msgs;
        int condcnt;
    exec sql include sqlca;
    exec sql end declare section;
    exec sql get diagnostics :num_msgs = number;
    for (condcnt=1; condcnt <= num_msgs; condcnt++)
    {
        exec sql get diagnostics exception :condcnt
            :sqlca = sqlca_info;
        printf("sqlcode is :%d\n\ message text:
            %s\n", sqlca.sqlcode,
            sqlca.sqlerrm.sqlerrmc);
    }
}
```

```
    }  
}
```

Writing routines to handle warnings and errors

A good strategy for handling errors and warnings in an Embedded SQL application is to write custom procedures to handle them, then install the procedures with the `whenever...call` statement.

The following example shows sample warning and error handling routines. For simplicity, both routines omit certain conditions that should normally be included. `warning_hndl` omits the code for `sqlwarn[1]`. `error_hndl` omits the code that handles Client-Library errors and operating system errors:

```
/* Declare the sqlca. */  
exec sql include sqlca;  
exec sql whenever sqlerror call error_handler();  
exec sql whenever sqlwarning call  
warning_handler();  
exec sql whenever not found continue;  
/*  
** void error_handler()  
**  
** Displays error codes and numbers from the sqlca  
**/  
void error_handler()  
{  
    fprintf(stderr,  
        "\n**sqlcode=(%d)", sqlca.sqlcode);  
    if (sqlca.sqlerrm.sqlerrml)  
    {  
        fprintf(stderr, "\n** ASE Error ");  
        fprintf(stderr, "\n** %s", sqlca.sqlerrm.sqlerrmc);  
    }  
    fprintf(stderr, "\n\n");  
    exit(ERREXIT);  
}  
/*  
** void warning_handler()  
**  
** Displays warning messages.  
**/  
void warning_handler()
```

```

{
    if (sqlca.sqlwarn[1] == 'W')
    {
        fprintf(stderr, "\n** Data truncated.\n");
    }

    if (sqlca.sqlwarn[3] == 'W')
    {
        fprintf(stderr, "\n** Insufficient
            host variables to store results.\n");
    }
    return;
}

```

Precompiler-detected errors

The Embedded SQL precompiler detects Embedded SQL errors at precompile time. The precompiler detects syntax errors such as missing semicolons and undeclared host variables in SQL statements. These are severe errors, so appropriate error messages are generated.

You can also have the precompiler check Transact-SQL syntax errors. Adaptive Server Enterprise parses Transact-SQL statements at precompile time if the appropriate precompiler command options are set. See the precompiler reference page in the *Open Client and Open Server Programmers Supplement* for your platform.

The precompiler does not detect the error in the following example, in which a table is created and data is selected from it. The error is that the host variables' datatypes do not match the columns retrieved. The precompiler does not detect the error because the table does not yet exist when the precompiler parses the statements:

```

exec sql begin declare section;
    CS_INT    var1;
    CS_CHAR   var2[20];
exec sql end declare section;

exec sql create table
    T1 (col1 int, col2 varchar(20));
....

```

```
exec sql select * from T1 into :var2, :var1;
```

Note that the error will be detected and reported at runtime.

Improving Performance with Persistent Binding

This chapter describes persistent binding and how it can improve performance. Persistent binding is a feature of Client-Library, the set of routines that executes Embedded SQL statements. Persistent binding improves a program's performance by enabling the Embedded SQL precompiler to create more efficient code.

Topic	Page
About persistent binding	102
Precompiler options for persistent binding	105
Overview of rules for persistent binding	106
Guidelines for using persistent binding	113
Notes on the binding of host variables	114

Persistent binding is optional: It takes effect if you request it when you precompile your program. Persistent binding benefits only certain types of Embedded SQL programs.

To understand this chapter, you should be familiar with host variables, cursors, dynamic SQL, and precompiler options. Refer to:

- Chapter 4, “Using Variables” for information about host variables.
- Chapter 6, “Using Transact-SQL Statements” for information about cursors.
- Chapter 7, “Using Dynamic SQL” for information about dynamic SQL.
- The *Open Client and Open Server Programmers Supplement* for information about precompiler options and about starting the precompiler.

You need not understand Client-Library to use persistent binding in Embedded SQL. However, understanding Client-Library's command structures, `ct_bind` routine, and `ct_fetch` routine can help you understand why persistent binding works as it does in Embedded SQL.

The general function of the command structures `ct_bind` and `ct_fetch` are described briefly in this chapter. For complete descriptions, refer to the *Open Client Client-Library/C Programmers Guide* and the *Open Client Client-Library/C Reference Manual*.

About persistent binding

To pass values to Adaptive Server and to store values from it, an Embedded SQL program uses host variables—C variables recognized by Embedded SQL. The program associates these variables with values on Adaptive Server. For example, the following `select` statement associates the host output variable `last` with a row value retrieved from Adaptive Server:

```
id = "998-72-3567";
exec sql select au_lname into :last
from authors where au_id = :id;
```

The statement passes its host input variable, `id`, to Adaptive Server and associates that variable with the server's `au_id` column.

The act of associating a statement's host variables with Adaptive Server values is called **binding**. The association itself is also called a binding. Host input variables use only input bindings; host output variable use only output bindings.

Binding governs which data a statement retrieves from the server. If a statement binds a host variable to the wrong server data, the statement will retrieve the wrong value for that host variable. However, unnecessary binding can slow a program's performance.

Embedded SQL lets you control how long bindings remain in effect—how long they “persist.” A binding that persists for more than one execution of a statement is called a **persistent binding**. Persistent bindings enable some Embedded SQL statements to execute faster, thereby improving a program's performance.

In Embedded SQL, each binding is made possible by a Client-Library **command structure**—a data structure that, among other things, defines the bindings of an Embedded SQL statement. For each Embedded SQL statement that executes, there is a corresponding command structure. A single command structure, however, can be used by more than one statement. In fact, when bindings persist from one Embedded SQL statement to another, they do so because the statements share a single command structure.

An Embedded SQL program's source code does not explicitly declare or allocate command structures. Instead, command structures are declared and allocated by the program's generated code.

When binding occurs

By default, binding occurs each time an Embedded SQL statement executes, using a host variable. When an Embedded SQL statement executes more than once, as in a loop, binding occurs at each execution. For example, in the following loop, each execution of the insert statement associates its host variables with the same Adaptive Server values. Yet, by default, binding occurs for each execution:

```
for (i = 1; i <= 3; i++)
{
    exec sql insert into titles (title_id, title)
    values (:bk_id, :bk_title);
    /*
    ** Binding occurs here at each execution.
    ** When a statement undergoes binding, all
    ** its host variables get bound.
    */
}
```

For most statements, bindings do not persist from one statement to the next, even if you request persistent binding. For example, the following insert statements, though identical and consecutive, share no bindings:

```
exec sql insert into titles (title_id, title)
    values (:bk_id, :bk_title);
/* Binding occurs for the first statement. */

exec sql insert into titles (title_id, title)
    values (:bk_id, :bk_title);
/* Binding occurs for the second statement. */

exec sql insert into titles (title_id, title)
    values (:bk_id, :bk_title);
/* Binding occurs for the third statement. */
```

For Embedded SQL statements that execute more than once—such as the insert statement in the preceding for loop—you can specify whether binding should occur only at the first execution or at each subsequent execution as well.

To control persistent binding, you use precompiler options to specify the binding behavior of all the statements in a file. Precompiler options do not let you control the binding behavior of individual statements. The precompiler options that control binding are explained later in this chapter.

Programs that can benefit from persistent binding

Not all Embedded SQL programs benefit from persistent binding. To find out whether persistent binding can benefit your program, answer the following questions:

- 1 Does your program contain at least one Embedded SQL statement that executes more than once?
- 2 If so, does that statement repeatedly use the same host variables to exchange values with Adaptive Server?

If you answered “yes” to both questions, your program can probably benefit from persistent binding. If you answered “no” to either question, persistent binding would not improve your program’s performance—unless you modify your program so that you can answer “yes” to both questions.

To maximize the benefit from persistent binding, your program should execute a single Embedded SQL statement repeatedly instead of executing two or more identical statements. For example, the following insert statement executes repeatedly:

```
for (i = 1; i <= 3; i++)
{
    exec sql insert into titles (title_id, title)
    values (:bk_id, :bk_title);
}
```

Although the insert statement in this example executes three times, its variables are bound only once. Because binding is not repeated, this example should run faster than a series of identical insert statements that execute only once.

Scope of persistent bindings

The scope of persistent bindings—how long they persist—differs depending on the type of statement and on the precompiler options in effect, as described later in this chapter. However, *bindings never persist beyond the lifetime of a connection*. When a program closes a connection, all bindings for statements issued and all command structures allocated over that connection are canceled.

Precompiler options for persistent binding

Two precompiler options control binding, the `-p` option and the `-b` option. These options affect only Embedded SQL statements that can use persistent binding. (Refer to Table 9-1 for a list of statements that cannot use persistent binding.)

The `-p` option

The `-p` option controls whether each statement has a **persistent command structure**—one that persists for all executions of a particular statement. Only statements with a persistent command structure can have persistent bindings for host variables. Thus, the `-p` option controls binding of host input variables, whose values are passed to Adaptive Server Enterprise. (In this chapter, information about “host input variables” also applies to other variables whose values are passed to Adaptive Server. Exceptions are noted in the text.)

The `-b` option

The `-b` option controls binding of host variables used in statements that retrieve result data from Adaptive Server Enterprise. When used in conjunction with the `-p` option, it controls binding of host variables in `select` and `exec` statements. When the `-b` option is used by itself, it can only control statements that fetch with a cursor.

Thus, generally, the `-b` option controls binding of host variables (output variables, result variables, status variables, indicator variables, and so on) whose values are passed from Adaptive Server Enterprise. (Information about “host output variables” also applies to any other variables whose values are output from Adaptive Server.) More precisely, the `-b` option controls whether binding occurs at each execution of Client-Library’s `ct_fetch` routine. (The `ct_fetch` routine retrieves a single row of data from Adaptive Server.)

Which option to use: `-p`, `-b`, or both

Most programs that can benefit from persistent bindings for input variables can also benefit from persistent bindings for output variables. In general, you should use both `-p` and `-b` options or use neither option.

Scope of the `-p` and `-b` precompiler options

The `-p` and `-b` options affect only the file being precompiled, unless that file declares a cursor. If the file declares a cursor, `-p` and `-b` affect all statements that use the cursor—even if those statements are in different source files of your program. The effect of `-p` and `-b` on files that use cursors is described in detail later in this chapter.

Overview of rules for persistent binding

The rules of persistent binding differ for different types of Embedded SQL statements. Specifically, the rules differ depending on whether a statement:

- Can use persistent binding
- Uses a cursor
- Is a dynamic SQL statement
- Is a fetch statement with the `rebind/norebind` clause

Statements that can use persistent binding

Most Embedded SQL statements can use persistent binding. However, Table 9-1 and Table 9-2 list Embedded SQL commands that cannot use persistent binding. All other Embedded SQL commands—including Transact-SQL commands—can use persistent binding for some or all host variables.

Whether a statement's bindings persist and how long they persist depends on the type of statement—particularly, on whether the statement uses a cursor.

Table 9-1: Embedded SQL commands that cannot use persistent binding

allocate descriptor	begin transaction
close	commit
connect	deallocate cursor
deallocate descriptor	deallocate prepare
describe input	describe output
disconnect	end transaction
execute	execute immediate
get descriptor	get diagnostics
open using descriptor	prepare
prepare transaction	rollback
set descriptor	set connection
set transaction diagnostics	

Table 9-2: Types of Embedded SQL commands that cannot use persistent binding

Commands that send text or image data to Adaptive Server with the -y option	Dynamic SQL commands that use a SQL descriptor or SQLDA for input to Adaptive Server Enterprise
---	---

Persistent binding in statements without a cursor

If an Embedded SQL statement can use persistent binding but does not use a cursor, you control the statement's bindings with the -p and -b options when precompiling the statement. Table 9-3 describes how these options affect a statement that uses no cursor.

Table 9-3: How -p and -b options affect statements with no cursors

Options used to precompile statement	Effect on statement's bindings
Neither -p nor -b	No bindings persist.
-p only	Only input bindings persist.
-b only	No bindings persist.
Both -p and -b	All bindings persist.

If the statement's bindings persist, they do so until your program closes the connection over which the statement executes. The bindings persist throughout all executions of the statement, even if other statements execute in the meantime. If the statement's bindings do not persist, binding occurs each time the statement executes.

Persistent binding in statements with a cursor

Before your program can use a cursor, you must declare it with the `declare cursor` command. A cursor's declaration governs the binding behavior of all statements that use the cursor—in all source files of your program. The reason for this control is that the command structure for a cursor's declaration is shared by all statements that use the cursor.

When a statement uses a cursor, the cursor's declaration—not the statement using the cursor—controls how long the statement's bindings persist. The bindings persist only if you use the `-b` and `-p` options when precompiling the file that declares the cursor. If you use these options, all statements that use the cursor have persistent bindings as specified by the options.

Strictly speaking, a cursor's declaration controls binding behavior only if the cursor is a **dynamic cursor**—a cursor for a dynamic SQL statement. In cursors for all other SQL statements (**static cursors**), the statement that most recently opened the cursor (open cursor) controls the binding behavior, not the statement that declares the cursor.

Note For a static cursor, the generated code for `open cursor` both declares and opens the cursor. For a dynamic cursor, the generated code for `open cursor` only opens the cursor.

Except for this difference, the binding rules for static cursors and dynamic cursors are the same. Unless you use a particular cursor in more than one source file of your program, the binding behavior of static cursors and dynamic cursors is the same.

In statements that use a cursor, bindings never persist after the cursor is deallocated, even if you use persistent binding. Also, deallocated cursors cannot be reopened. Declaring a new cursor with the name of a deallocated cursor does not reopen the deallocated cursor, nor does it retain bindings associated with that cursor. See the description of the `deallocate cursor` command in Chapter 10, “Embedded SQL Statements: Reference Pages.”

The following example shows how the `-b` and `-p` options affect a cursor—in this example, *curs1*. The `fetch` statement in the example contains host variables. The paragraphs following the example describes how the `-b` and `-p` options affect the bindings of these host variables.

```
#include <stdio.h>
int SQLCODE;

void
main()
{
    exec sql begin declare section;
        char title[100], pub_id[8];
    exec sql end declare section;

    exec sql connect "sa";

    exec sql use pubs2;
        /*
        ** The options used to precompile a cursor's declaration
        ** control whether host variables persist in statements,
        ** such as FETCH, that use the cursor.
        */
    exec sql declare curs1 cursor for select title, pub_id from
        titles;
    exec sql open curs1;

    while (SQLCODE == 0)
    {
        /* If the declaration of curs1 was precompiled without
        ** the -b option, rebind the FETCH statement's variables
        ** each time the statement repeats. Otherwise, bind only
        ** the first time, and let the bindings persist for
        ** subsequent repetitions.

```

```
*/
exec sql fetch curs1 into :title, :pub_id;
      printf("%s, %s\n", title, pub_id);
}
/* If the declaration of curs1 was precompiled without
** the -p option, cancel the bindings of the FETCH
** statement's variables when curs1 is closed.
** Otherwise, let the bindings persist until the
** program deallocates curs1 or, as here, until the
** program ends.
*/
exec sql close curs1;
exec sql disconnect CURRENT;

exit(0);
```

Preventing persistent binding for all cursor host variables

If you omit both the `-b` and `-p` options when precompiling the above example, no bindings persist. Instead, the generated code binds the host variables each time the fetch statement executes, regardless of whether the variable is input to Adaptive Server Enterprise or output from it.

Requesting persistent binding for all cursor host variables

If you use both the `-b` and `-p` options when precompiling the preceding example, the generated code binds the host variables of the fetch statement only the first time that the statement executes. Unlike other Embedded SQL statements (as described in “When binding occurs” on page 103), it does not matter whether there are one or more identical fetch statements in a series, or a simple fetch statement executed in a loop. Using both options together causes the bindings to persist even when the program closes the cursor; the host variables do not need to be rebound when the cursor is reopened. The bindings persist until the program deallocates the cursor—typically, with the `deallocate cursor` or `disconnect` statement.

Requesting persistent binding for cursor output variables only

If you use `-b` but omit `-p` when precompiling the preceding example, the generated code binds the host output variables of the fetch statement only once—the first time that the statement executes. (More precisely, the host variables get bound only if they are not bound already.) The bindings persist for all subsequent executions of the statement, until the program closes *cursor1*. They persist because you used the `-b` option. Because you omitted the `-p` option, bindings for host input variables do not persist.

If your program closes *cursor1* and then reopens it, all bindings for host variables related to *cursor1* are canceled. Any host input variables and host output variables are re-bound when the cursor is reopened. They persist until your program closes the cursor again.

Requesting persistent binding for cursor input variables only

The preceding example showed how the `-b` and `-p` options affect statements that use host variables with a cursor. The example's only host variables were host output variables. The following code is an example that shows how the `-b` and `-p` options affect statements that use host input variables with a cursor in this case, a dynamic cursor named *dyn_cursor1*.

The open statement in the following example contains a host input variable, *min_price*. The following sections describe how the `-b` and `-p` options affect the bindings of this host input variable.

```
#include <stdio.h>
long SQLCODE = 0;

void main()
{
    int i = 0;
    exec sql begin declare section;
        CS_CHAR          sql_string[200];
        CS_FLOAT         min_price;
        CS_CHAR          book_title[200];
    exec sql end declare section;
    exec sql connect "sa";
    exec sql use pubs2;
    strcpy(sql_string,
        "select title from titles where price > ?");
    exec sql prepare sel_stmt from :sql_string;

    /* The options used to precompile a cursor's declaration
    ** control whether host variables persist in statements,
```

```
** such as OPEN, that use the cursor.
*/
exec sql declare dyn_curs1 cursor for sel_stmt;
min_price = 10.00;
/* If the declaration of dyn_curs1 was precompiled
** without -p, bind the OPEN statement's input variable
** (min_price) each time the statement repeats. Otherwise,
** bind only the first time, letting the binding persist
** until dyn_curs1 is deallocated.
*/

for (i = 10; i <= 21; ++i)
{
    min_price = min_price + 1.00;
    exec sql open dyn_curs1 using :min_price;
    while (SQLCODE != 100)
    {
        exec sql fetch dyn_curs1 into :book_title;
        if (SQLCODE != 100) printf("%s\n", book_title);
    }
    printf("\n");
    exec sql close dyn_curs1;
}
exec sql deallocate cursor dyn_curs1;
exec sql disconnect CURRENT;
exit(0);
}
```

If you use `-p` but omit `-b` when precompiling the preceding example, the generated code binds `min_price` only once—the first time that the open statement executes. The binding persists because you used the `-p` option, which controls host input variables.

The binding for `min_price` persists throughout all subsequent iterations of the statement, until the program deallocates `dyn_curs1`. The binding persists even if your program closes `dyn_curs1` and then reopens it.

Persistent binding, cursors, and multiple source files

In the preceding example, the declaration of the cursor `dyn_curs1` controls whether associated host variables persist. For this reason, the host variables in the fetch statement would bind as described in the example, even if the fetch statement were precompiled in a separate source file.

Persistent binding and cursor *fetch* statements

The Embedded SQL *fetch* command has an optional *rebind/norebind* clause that controls whether bindings persist in a particular *fetch* statement. This clause is useful if you need to override the precompiler options that you specified for a file. The *rebind/norebind* clause affects only the statement in which it appears. Bindings for other statements—including other *fetch* statements—are not affected.

If a *fetch* statement omits the *rebind/norebind* clause, the statement obeys the same binding rules as do other types of statements that use the cursor in question.

If a *fetch* statement contains the keyword *rebind*, bindings for host variables in the statement do not persist. Instead, they get rebound each time the statement executes—regardless of whether the *-b* option was used to precompile the declaration of the statement's cursor.

If a *fetch* statement contains the keyword *norebind* but is precompiled with the *-b* option, the keyword has no effect.

Guidelines for using persistent binding

Here are guidelines, tips, and reminders to help you use persistent binding correctly:

- A program benefits from persistent binding only if it meets both of these criteria:
 - It contains at least one Embedded SQL statement that executes more than once, and
 - That statement uses the same host variables repeatedly to exchange values with Adaptive Server.
- The *-p* and *-b* options affect only the file being precompiled, unless that file declares a cursor. If the file declares a cursor, *-p* and *-b* affect all statements that use the cursor. In general, you should use both the *-p* and *-b* options or use neither. If your program consists of more than one Embedded SQL source file, you should generally use the same combination of the *-p* and *-b* options to precompile all the files.

Generally, if you use the same cursor in more than one source file of a program, use the same combination of the `-p` and `-b` options when precompiling those files. Otherwise, you will need to understand exactly how different combinations of the options can change which data a statement sends or retrieves.

- A program that uses persistent binding should, where practical, execute a single Embedded SQL statement repeatedly instead of executing two or more identical statements once each.
- The rules controlling a statement's bindings differ depending on whether the statement:
 - Can use persistent binding
 - Uses a cursor
 - Is a dynamic SQL statement
 - Is a fetch statement with the `rebind/norebind` clause
- Bindings never persist beyond the lifetime of a connection. In statements that use a cursor, bindings never persist after the cursor is deallocated.
- A dynamic cursor's declaration controls the binding behavior of all statements that use the cursor. For a static cursor, the statement that most recently opened the cursor exerts this control. A program should open a static cursor only in the source file that declares it.

Notes on the binding of host variables

The following describes the behavior of subscripted array host variables and of host variables when used in repeated executions.

Subscripted arrays

If you use `-p` or `-b` and bind a subscripted array host variable (input or output), the subscript is ignored after the first execution of the statement, because the actual address of the specified array element is bound. For example:

```
exec sql begin declare section;
int row;
int int_table[3] = {
```

```

    10,
    20,
    30,
};
char *string_table[3] = {
    "how",
    "are",
    "you",
};
exec sql end declare section;
for (row=0; row < 3; row++)
{
    EXEC SQL insert into ... values (:row, :int_table[row],
        :string_table[row]);
    /*
    ** If this statement is precompiled with -p, only
    ** int_table[0] and string_table[0] will be bound and
    ** inserted each time.
    ** The same thing applies to output variables
    ** At this time, NO warnings are issued to detect this.
    */
}

```

To solve this, you can choose among the following solutions:

- Do not use persistent binds when subscripted arrays are used, since you *do* want a rebind (**table[0]* is not the same as **table[1]* at the next iteration).
- If persistent binds must be used, use an intermediate variable that holds the current value. This method allows persistent binding without errors. However, copying the data creates overhead. Using the above example:

```

exec sql begin declare section;
char    bind_str[80];
int bind_int_variable;
exec sql end declare section;
for (row=0; row < 3; row++)
{
    /*
    ** Must copy the contents- pointer assignment does
    ** not suffice host var 'row' is not a subscripted
    ** array, so it can remain the same.
    */
    memcpy(bind_str, string_table[row], 80);
    bind_int_variable = int_table[row];
    EXEC SQL insert into ... values (:row,
        :bind_int_variable,
        :bind_str);
}

```

```
}
```

Note No register variables can be used with persistent binding.

Scope of host variables

When host variables remain bound from one execution to the next, you must ensure that they remain in scope. Particular care must be taken when automatic variables such as stack variables are used.

When a possibly problematic situation can be detected by the precompiler, a warning is issued. Whether a host variable remains in scope or not will also depend on the overall program logic.

For example:

```
/*
** a function called by main()
*/
CS_VOID insert(insert_row)
exec sql begin declare section;
int insert_row; /* row will go out of scope once exit
                ** function*/
exec sql end declare section;
{
  /*
    ** id is a stack variable which will go out of scope
    ** once we exit the function insert()
    ** it is not likely to be at the same address at the
    ** next call to this function, so if it is bound as
    ** an input variable, there will be errors.
    */
  exec sql begin declare section;
  int id;
  exec sql end declare section;
  exec sql insert values(:row,:id);
}
int fetched_row; /* this variable can be safely bound with
                 ** persistence */
main()
{
  exec sql begin declare section;
  /*
    ** This variable will go out of scope when the program
```

```
        ** exits main, which is not a problem.
    */
int row;
/*
    ** This variable is a pointer, thus it does not
    ** necessarily pose problems, depending on the scope
    ** of the data it is pointing to.
    */
char *pointer;
exec sql end declare section;
for (row = 0; row < 10; row++)
{
    insert(row);
}
}
```


Embedded SQL Statements: Reference Pages

This chapter consists of a reference page for each Embedded SQL statement that either does not exist in Transact-SQL, or works differently from how it does in Transact-SQL. Refer to the *Adaptive Server Enterprise Transact-SQL Users Guide* for descriptions of all other Transact-SQL statements that are valid in Embedded SQL.

Command statements	Page
allocate descriptor	121
begin declare section	122
begin transaction	123
close	124
commit	126
connect	127
deallocate cursor	129
deallocate descriptor	131
deallocate prepare	131
declare cursor (dynamic)	132
declare cursor (static)	133
declare cursor (stored procedure)	135
declare scrollable cursor	136
delete (positioned cursor)	138
delete (searched)	139
describe input (SQL descriptor)	141
describe input (SQLDA)	142
describe output (SQL descriptor)	143
describe output (SQLDA)	144
disconnect	145
exec	147
exec sql	150
execute	152
execute immediate	154
exit	155

Command statements	Page
fetch	155
fetch scrollable cursor	158
get descriptor	159
get diagnostics	161
include "filename"	162
include sqlca	164
include sqllda	165
initialize_application	165
open (dynamic cursor)	167
open (static cursor)	168
prepare	170
rollback	172
select	172
set connection	174
set descriptor	175
thread exit	177
update	177
whenever	179

Except for print, readtext and writetext, all Transact-SQL statements can be used in Embedded SQL, though the syntax of some statements differs as described in this chapter.

The reference pages in this chapter are arranged alphabetically. Each statement's reference page:

- Briefly states what the statement does
- Describes the statement's syntax
- Explains the statement's keywords and options
- Comments on the statement's proper use
- Lists related statements, if any
- Demonstrates the statement's use in a brief example

allocate descriptor

Description	Allocates a SQL descriptor.
Syntax	<code>exec sql allocate descriptor <i>descriptor_name</i> [with max [<i>host_variable</i> <i>integer_literal</i>]];</code>
Parameters	<p><i>descriptor_name</i> The name of the SQL descriptor that will contain information about the dynamic parameter markers in a prepared statement.</p> <p>with max The maximum number of columns in the SQL descriptor.</p> <p><i>host_variable</i> An integer host variable defined in a declare section.</p> <p><i>integer_literal</i> A numeric value representing the size, in number of occurrences, of the SQL descriptor.</p>
Examples	<pre>exec sql begin declare section; CS_INT type; CS_INT numcols, colnum; exec sql end declare section; ... exec sql allocate descriptor big_desc with max 1000; exec sql prepare dynstmt from "select * from huge_table"; exec sql execute dynstmt into sql descriptor big_desc; exec sql get descriptor :numcols = count; for (colnum = 1; colnum <= numcols; colnum++) { exec sql get descriptor big_desc :type = type; ... } exec sql deallocate descriptor big_desc; ...</pre>
Usage	<ul style="list-style-type: none"> • The allocate descriptor command specifies the number of item descriptor areas that Adaptive Server Enterprise allocates. • You can allocate any number of SQL descriptors. • When a SQL descriptor is allocated, its fields are undefined. • If you try to allocate a SQL descriptor that is already allocated, an error occurs.

- If you do not specify a value for the with max clause, one item descriptor is assigned.
- When a SQL descriptor is allocated, the value of each of its fields is undefined.

See also `deallocate descriptor`, `get descriptor`, `set descriptor`

begin declare section

Description Begins a declare section, which declares host language variables used in an Embedded SQL source file.

Syntax `exec sql begin declare section;`
`host_variable_declaration;`
...
`exec sql end declare section;`

Parameters *host_variable_declaration*
The declaration of one or more host language variables.

Examples

```
exec sql begin declare section;
        CS_CHAR    name (80) ;
        CS_INT     value;
exec sql end declare section;
```

Usage

- A declare section must end with the Embedded SQL statement `end declare section`.
- A source file can have any number of declare sections.
- A declare section can be placed anywhere that variables can be declared. The declare section that declares a variable must precede any statement that references the variable.
- Variable declarations in a declare section must conform to the rules of the host language.
- Nested structures are valid in a declare section; arrays of structures are not.
- A declare section can contain any number of Embedded SQL include statements.
- In Embedded SQL/C routines, the Client-Library datatypes defined in *cspublic.h* can be used in declare sections.

- In C routines, you can declare two-dimensional arrays of characters but only one-dimensional arrays of other datatypes.
- When processing declare sections, the Embedded SQL precompiler ignores C preprocessor macros and #include statements. When processing Embedded SQL include statements within a declare section, the Embedded SQL precompiler treats the contents of the included file as though they had been entered directly into the file being precompiled.

See also `exec sql include "filename"`

begin transaction

Description Marks the starting point of an unchained transaction.

Syntax `exec sql [at connection_name]
begin {transaction | tran} [transaction_name];`

Parameters `transaction | tran`
The keywords `transaction` and `tran` are interchangeable.

transaction_name

The name that you are assigning to this transaction. The name must conform to the rules for Transact-SQL identifiers.

Examples

```
/*
** Use explicit transactions to
** synchronize tables on two servers
*/
exec sql begin declare section;
    char        title_id[7];
    int         num_sold;
exec sql end declare section;
    long        sqlcode;
    ...

exec sql whenever sqlerror goto abort_tran;
try_update;

exec sql at connect1 begin transaction;

exec sql at connect2 begin transaction;
exec sql at connect1 select sum(qty)
    into :num_sold
    from salesdetail
    where title_id = :title_id;
```

```
exec sql at connect2 update current_sales
    set num_sold = :num_sold
    where title_id = :title_id;
exec sql at connect2 commit transaction;
exec sql at connect1 commit transaction;
if (sqlcode != 0)
    printf("oops, should have used 2-phase
        commit\n");
return;
abort_tran:
exec sql whenever sqlerror continue:
exec sql at connect2 rollback transaction;
exec sql at connect1 rollback transaction;
goto try_update;
```

Usage

- This reference page describes aspects of the Transact-SQL begin transaction statement that differ when used with Embedded SQL. See the *Adaptive Server Enterprise Reference Manual*.
- The begin transaction statement is valid only in unchained transaction mode. In chained transaction mode, you cannot explicitly mark the starting point of a transaction.
- When nesting transactions, assign a transaction name only to the outermost begin transaction statement and its corresponding commit transaction or rollback transaction statement.
- Unless you set the database option `ddl in tran`, Adaptive Server Enterprise does not allow the following statements inside an unchained transaction: create database, create table, create index, create view, drop, select into table_name, grant, revoke, alter database, alter table, truncate table, update statistics, load database, load transaction, and disk init.
- A transaction includes only statements that execute on the connection that is current when the transaction begins.
- Remote procedures execute independently of any transaction in which they are included.

See also

commit transaction, commit work, rollback transaction, rollback work

close

Description

Closes an open cursor.

Syntax	<code>exec sql [at <i>connection_name</i>] close <i>cursor_name</i>;</code>
Parameters	<p><i>cursor_name</i></p> <p>The name of the cursor to be closed; that is, the name that you assigned when declaring the cursor.</p>
Examples	<pre> long SQLCODE; exec sql begin declare section; CS_CHAR mlname[40]; CS_CHAR mfname[20]; CS_CHAR phone[12]; exec sql end declare section; exec sql declare author_list cursor for select au_lname, au_fname, phone from authors; exec sql open author_list; while (SQLCODE == 0) { exec sql fetch author_list into :mlname, :mfname, :mphone; if (SQLCODE != 100) printf("%s, %s, %s\n", mlname, mfname, mphone); } exec sql close author_list; </pre>
Usage	<ul style="list-style-type: none"> • The close statement closes an open cursor. Unfetched rows are canceled. • Reopening a closed cursor executes the associated query again, positioning the cursor pointer before the first row of the result set. • A cursor must be closed before it is reopened. • Attempting to close a cursor that is not open causes a runtime error. • The commit transaction, rollback transaction, commit work, and rollback work statements close a cursor automatically unless you set a precompiler option to disable the feature. • Closing and then reopening a cursor lets your program see any changes in the tables from which the cursor retrieves rows.
See also	<code>declare cursor</code> , <code>fetch</code> , <code>open</code> , <code>prepare</code>

commit

Description	Ends a transaction, preserving changes made to the database during the transaction.
Syntax	<pre>exec sql [at <i>connection_name</i>] commit [transaction tran work] [<i>transaction_name</i>];</pre>
Parameters	<p>transaction trans work</p> <p>The keywords transaction, trans, and work are interchangeable in the rollback statement, except that only work is ANSI-compliant.</p> <p><i>transaction_name</i></p> <p>A name assigned to the transaction.</p>
Examples	<pre>/* ** Using chained transaction mode, ** synchronize tables on two servers */ exec sql begin declare section; char title_id[7]; int num_sold; exec sql end declare section; long SQLCODE; ... try_update: exec sql whenever sqlerror goto abort_tran; exec sql at connect1 select sum(qty) into :num_sold from salesdetail where title_id = :title_id; exec sql at connect2 update current_sales set num_sold = :num_sold where title_id = :title_id; exec sql at connect2 commit work; exec sql at connect1 commit work; return; abort_tran: printf("oops, should have used 2-phase commit\n"); exec sql whenever sqlerror continue; exec sql at connect2 rollback work; exec sql at connect1 rollback work; goto try_update;</pre>

Usage	<ul style="list-style-type: none"> • This reference page mainly describes aspects of the Transact-SQL commit statement that differ when used with Embedded SQL. See the <i>Adaptive Server Enterprise Reference Manual</i>. • Transaction names must conform to the Transact-SQL rules for identifiers. Transaction names are a Transact-SQL extension: they cannot be used with the ANSI-compliant keyword work. • When nesting transactions, assign a transaction name only to the outermost begin transaction statement and its corresponding commit transaction or rollback transaction statement.
See also	begin transaction, commit work, rollback transaction, rollback work

connect

Description	Creates a connection to Adaptive Server Enterprise.
Syntax	<pre>exec sql connect <i>user_name</i> [identified by <i>password</i>] [at <i>connection_name</i>] [using <i>server_name</i>];</pre>
Parameters	<p><i>user_name</i> The user name to be used when logging in to Adaptive Server Enterprise.</p> <p><i>password</i> The password to use to log in to Adaptive Server Enterprise.</p> <p><i>connection_name</i> A name that you choose to uniquely identify the Adaptive Server Enterprise connection.</p> <p><i>server_name</i> The server name of the Adaptive Server Enterprise to which you are connecting.</p>
Examples	<pre>exec sql begin declare section; CS_CHAR user [32]; CS_CHAR password [32]; CS_CHAR server [90]; CS_CHAR conname [20]; exec sql end declare section; strcpy(user, "mylogin"); strcpy(password, "mypass"); strcpy(server, "YOURSERVER");</pre>

```
strcpy(conname, "con_one");  
exec sql connect :user identified by :password  
using :server at :conname;
```

Usage

- In every Embedded SQL program, the connect statement must be executed before any other executable SQL statement except allocate descriptor.
- If a program uses both C and COBOL languages, the first connect statement must be issued from a COBOL program.
- If a program has multiple connections, only one can be unnamed, and will be the default connection.
- If an Embedded SQL statement does not have an *at connection_name* clause to direct it to a specific named connection, the statement is executed on the current connection.
- To specify a null password, omit the identified by clause or use an empty string.
- If the connect statement does not specify a Adaptive Server Enterprise, the server named by the DSQUERY environment variable or logical name is used. If DSQUERY is not defined, the default server is SYBASE.
- Client-Library looks up the server name in the interfaces file located in the directory specified by the SYBASE environment variable or logical name.
- The Adaptive Server Enterprise connection ends when the Embedded SQL program exits or issues a disconnect statement.
- Opening a new connection, named or unnamed, results in the new connection becoming the current connection.
- A program that requires multiple Adaptive Server Enterprise login names can have a connection for each login account.
- By connecting to more than one server, a program can simultaneously access data stored on different servers.
- A single program can have multiple connections to a single server or multiple connections to different servers.
- Table 10-1 shows how a connection is named:

Table 10-1: How a connection is named

If this clause is used	But without	Then, the Connection Name is
at <i>connection_name</i>		<i>connection_name</i>
using <i>server_name</i>	at	<i>server_name</i>
None		DEFAULT

See also `at connection_name`, `exec sql`, `disconnect`, `set connection`

deallocate cursor

Description	Deallocates a cursor for a static SQL statement or for a dynamic SQL statement.
Syntax	<code>exec sql [at <i>connection_name</i>] deallocate cursor <i>cursor_name</i>;</code>
Parameters	<p><i>cursor_name</i></p> <p>The name of the cursor to be deallocated. The <i>cursor_name</i> must be a character string enclosed in double quotation marks or in no quotation marks—for example “<i>my_cursor</i>” or <i>my_cursor</i>. It cannot be a host variable.</p>

Examples

```

exec sql include sqlca;
main()
{
exec sql begin declare section;
    CS_CHAR title[80];
    CS_SMALLINT i_title;
exec sql end declare section;
exec sql whenever sqlerror call error_handler();
exec sql whenever sqlwarning call error_handler();
exec sql whenever not found continue;
exec sql connect "sa";
exec sql use pubs2;
exec sql declare title_list cursor for select title from titles;

exec sql open title_list;
for (;;)
{
    exec sql fetch title_list into :title :i_title;
    if (sqlca.sqlcode == 100) break;

```

```
        if (i_title == -1) printf("Title is NULL.\n");

        printf("Title: %s\n", title);
    }
    exec sql close title_list;
    exec sql deallocate cursor title_list;
    exec sql disconnect all;
    exit(0);
}
error_handler()
{
    printf("%d\n%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
    exec sql deallocate cursor title_list;
    exec sql disconnect all;
    exit(-1);
}
```

Usage

- Deallocating a cursor releases all resources allocated to the cursor. In particular, deallocate cursor drops the Client-Library command handle and CS_COMMAND structure associated with the cursor.
- A static cursor can be deallocated at any time after it is opened. A dynamic cursor can be deallocated at any time after it is declared.
- If *cursor_name* is open, deallocate cursor closes it and then deallocates it.
- You cannot reference a deallocated cursor, nor can you reopen it. If you try, an error occurs.
- You can declare a new cursor having the same name as that of a deallocated cursor. Opening a cursor with the same name as a deallocated cursor is not the same as reopening the deallocated cursor. Other than the name, the new cursor shares nothing with the deallocated cursor.
- Declaring a new cursor with the same name as that of a deallocated cursor can cause the precompiler to generate a warning message.
- The deallocate cursor statement is a Sybase extension; it is not defined in the SQL standard.

Note If you are using persistent binding in your Embedded SQL program, use the deallocate cursor statement carefully. Needlessly deallocating cursors can negate the advantage of persistent binding.

See also

close cursor, declare cursor, open (static cursor)

deallocate descriptor

Description	Deallocates a SQL descriptor.
Syntax	<code>exec sql deallocate descriptor <i>descriptor_name</i>;</code>
Parameters	<i>descriptor_name</i> The name of the SQL descriptor that contains information about the dynamic parameter markers or return values in a prepared statement.
Examples	<pre> exec sql begin declare section; CS_INT numcols, colnum; exec sql end declare section; ... exec sql allocate descriptor big_desc with max 1000; exec sql prepare dynstmt from "select * from huge_table"; exec sql execute dynstmt into sql descriptor big_desc; exec sql get descriptor :numcols = count; for (colnum = 1; colnum <= numcols; colnum++) { exec sql get descriptor big_desc ... } exec sql deallocate descriptor big_desc; ... </pre>
Usage	If you attempt to deallocate a SQL descriptor that has not been allocated, an error occurs.
See also	allocate descriptor

deallocate prepare

Description	Deallocates a dynamic SQL statement that was prepared in a prepare statement.
Syntax	<code>exec sql [at <i>connection_name</i>] deallocate prepare <i>statement_name</i>;</code>
Parameters	<i>statement_name</i> The identifier assigned to the dynamic SQL statement when the statement was prepared.

Examples

```
exec sql begin declare section;
      CS_CHAR          sqlstmt[100];
      exec sql end declare section;
strcpy(sqlstmt, "select * from publishers");
exec sql prepare make_work from :sqlstmt;
exec sql declare make_work_cursor cursor for
      make_work;
exec sql deallocate prepare make_work;
```

Usage

- A statement must be prepared before it is deallocated. Attempting to deallocate a statement that has not been prepared results in an error.
- *statement_name* must uniquely identify a statement buffer and must conform to the SQL identifier rules for naming variables. *statement_name* can be either a literal or a character array host variable.
- The deallocate prepare statement closes and deallocates any dynamic cursors declared for *statement_name*.

Warning! If you are using persistent binds in your embedded SQL program, use the deallocate prepare statement carefully. Needlessly deallocating prepared statements can negate the advantage of persistent binds.

See also

declare cursor (dynamic), execute, execute immediate, prepare

declare cursor (dynamic)

Description

Declares a cursor for processing multiple rows returned by a prepared dynamic select statement.

Syntax

```
exec sql [at connection_name]
declare cursor_name
cursor for prepped_statement_name;
```

Parameters

cursor_name

The cursor's name, used to reference the cursor in open, fetch, and close statements. A cursor's name must be unique on each connection and must have no more than 255 characters.

prepped_statement_name

The name (specified in a previous prepare statement) that represents the select statement to be executed.

Examples

```
exec sql begin declare section;
```

```

CS_CHAR          sqlstmt[100];
exec sql end declare section;
strcpy(sqlstmt, "select * from publishers");
exec sql prepare make_work from :sqlstmt;
exec sql declare make_work_cursor cursor for
    make_work;
exec sql deallocate prepare make_work;

```

Usage

- The *prepped_statement_name* must not have a compute clause.
- The *cursor_name* must be declared on the connection where *prepped_statement_name* was prepared.
- The dynamic declare cursor statement is an executable statement, whereas the static declare cursor statement is simply a declaration. The dynamic declare statement must be located where the host language allows executable statements and the program should check return codes (SQLCODE, SQLCA, or SQLSTATE).
- The for update and read only clauses for a dynamic cursor are not part of the declare cursor statement; however, they should be included in the prepared statement's select query.

See also

close, connect, fetch, open, prepare

declare cursor (static)

Description Declares a cursor for processing multiple rows returned by a select statement.

Syntax

```

exec sql declare cursor_name
    cursor for select_statement
    [for update [of col_name_1 [, col_name_n]...]]
    for read only];

```

Parameters

cursor_name

The cursor's name, used to reference the cursor in open, fetch, and close statements. A cursor's name must be unique on each connection and must have no more than 255 characters.

select_statement

The Transact-SQL select statement to be executed when the cursor is opened. See the description of the select statement in the *Adaptive Server Enterprise Reference Manual*.

for update

Specifies that the cursor's result list can be updated. (To update the result list, you use the update statement.)

of *col_name_1*

The name of the first column to be updated.

of *col_name_n*

The name of the *n*th column to be updated.

for read only

Specifies that the cursor's result list cannot be updated.

Examples

```
main()
{
    exec sql begin declare section;
        CS_CHAR          b_titleid[TIDSIZE+1];
        CS_CHAR          b_title[65];
        CS_CHAR          b_type[TYPESIZE+1];
    exec sql end declare section;
        long            SQLCODE;
    exec sql connect "sa";
    exec sql use pubs2;
    exec sql declare titlelist cursor for
        select title_id, substring(title,1,64)
        from titles where type like :b_type;
    strcpy(b_type, "business");
    exec sql open titlelist;
    for (;;)
    {
        exec sql fetch titlelist into :b_titleid,
            :b_title;
        if (SQLCODE == 100)
            break;
        printf("  %-8s %s\n", b_titleid, b_title);
    }
    exec sql close titlelist;
    exec sql disconnect all;
}
```

Usage

- The Embedded SQL precompiler generates no code for the declare cursor statement.
- The *select_statement* does not execute until your program opens the cursor by using the open cursor statement.

- The syntax of the *select_statement* is identical to that shown in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use the compute clause in Embedded SQL.
- The *select_statement* can contain host variables. The values of the host variables are substituted when your program opens the cursor.
- If you omit either the for update or read only clause, Adaptive Server Enterprise determines whether the cursor is updatable.

See also close, connect, deallocate cursor, declare cursor (stored procedure),
declare cursor (dynamic), fetch, open, update

declare cursor (stored procedure)

Description	Declares a cursor for a stored procedure.
Syntax	<pre>exec sql declare <i>cursor_name</i> cursor for execute <i>procedure_name</i> ([[<i>@param_name</i> =]:<i>host_var</i>] [, [<i>@param_name</i> =]:<i>host_var</i>]...)</pre>
Parameters	<p><i>cursor_name</i> The cursor's name, used to reference the cursor in open, fetch, and close statements. A cursor's name must be unique on each connection and must have no more than 255 characters.</p> <p><i>procedure_name</i> The name of the stored procedure to be executed.</p> <p><i>param_name</i> The name of a parameter in the stored procedure.</p> <p><i>host_var</i> The name of a host variable to be passed as a parameter value.</p>
Examples	<pre>main() { exec sql begin declare section; CS_CHAR b_titleid[7]; CS_CHAR b_title[65]; CS_CHAR b_type[13]; exec sql end declare section; long SQLCODE; exec sql connect "sa"; exec sql use pubs2;</pre>

```
exec sql
  create procedure p_titles
    (@p_type varchar(30)) as
  select title_id, substring(title,1,64)
  from titles
  where type like @p_type;
exec sql declare titlelist cursor for
  execute p_titles (:b_type);
strcpy(b_type, "business");
exec sql open titlelist;
for (;;)
{
  exec sql fetch titlelist into :b_titleid,
    :b_title;
  if (SQLCODE == 100)
    break;
  printf("   %-8s %s\n", b_titleid, b_title);
}
exec sql close titlelist;
exec sql disconnect all;
}
```

- Usage
- *procedure_name* must consist of only one select statement.
 - It is not possible to retrieve output parameter values from a stored procedure executed using a cursor.
 - It is not possible to retrieve the return status value of a stored procedure executed using a cursor.

See also close, deallocate cursor, declare cursor (static), declare cursor (dynamic), fetch, open, update

declare scrollable cursor

Description Declare a cursor for each select statement that returns rows of data. You must declare the cursor before using it, and you cannot declare it within a declare section.

Syntax exec sql declare *cursor_name* [*cursor sensitivity*]
 [*cursor scrollability*] cursor for *select_statement* ;

Parameters *cursor_name*
 Identifies the cursor.

cursor sensitivity

Specifies the sensitivity of the cursor.

cursor scrollability

Specifies the scrollability of the cursor.

select_statement

A select statement that can return multiple rows of data. The syntax for select is the same as described in the *Adaptive Server Enterprise Reference Manual*, except that you cannot use into or compute clauses.

Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char    username[30];
char    password[30];
char    a_type[TITLE_STRING+1];
EXEC SQL END DECLARE SECTION;

.....

/*
** Declare an insensitive scrollable cursor against the
** titles table. Open the cursor.
*/
EXEC SQL DECLARE typelist INSENSITIVE SCROLL CURSOR FOR
SELECT DISTINCT title FROM titles;

EXEC SQL OPEN typelist;
```

Usage

- *cursor_name* must be unique and have a maximum of 255 characters.
- *cursor_name* must begin with a letter of the alphabet or with the symbols “#” or “_”.
- If *cursor sensitivity* is declared as *semi_sensitive*, scrollability is implied. The cursor is *semi_sensitive*, scrollable, and read-only.
- If *cursor sensitivity* is declared as *insensitive*, the cursor is insensitive. Scrollability is determined by specifying *SCROLL* in the declare part. If *SCROLL* is omitted or *NOSCROLL* is specified, the cursor is insensitive only and non-scrollable. It is also read-only.
- If cursor sensitivity is not specified, the cursor is non-scrollable and read-only.
- If *cursor scrollability* is specified as *scroll* in the declare statement and sensitivity is not specified, the cursor is insensitive and scrollable. It is also read-only.

- If the SCROLL option is omitted or NOScroll is specified in *cursor scrollability*, the cursor is non-scrollable and read-only.
- If cursor scrollability is not specified, the cursor is non-scrollable and read-only.

See also [fetch scrollable cursor](#)

delete (positioned cursor)

Description Removes, from a table, the row indicated by the current cursor position for an open cursor.

Syntax `exec sql [at connection_name] delete
[from] table_name
where current of cursor_name;`

Parameters *table_name*
The name of the table from which the row will be deleted.

where current of cursor_name
Causes Adaptive Server Enterprise to delete the row of the table indicated by the current cursor position for the cursor *cursor_name*.

Examples

```
exec sql include sqlca;
main()
{
    char answer[1];
    exec sql begin declare section;
        CS_CHAR disc_type[40];
        CS_CHAR store_id[5];
        CS_SMALLINT ind_store_id;
    exec sql end declare section;
    exec sql connect "sa";
    exec sql use pubs2;
    exec sql declare purge_cursor cursor for
        select discounttype, stor_id
        from discounts;
    exec sql open purge_cursor;
    exec sql whenever not found goto alldone;
    while (1)
    {
        exec sql fetch purge_cursor into :disc_type,
        :store_id
            :ind_store_id;
```

```

        if (ind_store_id != -1)
        {
            printf("%s, %s\n", disc_type, store_id);
            printf("Delete Discount Record? (y/n) >");
            gets(answer);
            if (strncmp(answer, "y", 1) == 0)
            {
                exec sql delete from discounts where
                    current of purge_cursor;
            }
        }
    }
}
/*
** No changes will be committed to the database because
** this program does not contain an "exec sql commit
work;"
** statement. The changes will be rolled back when the
** user disconnects.
*/
alldone:
    exec sql close purge_cursor;
    exec sql disconnect all;
}

```

Usage

- This reference page mainly describes aspects of the Transact-SQL delete statement that differ when used with Embedded SQL. See the *Adaptive Server Enterprise Reference Manual*.
- This form of the delete statement must execute on the connection where the cursor *cursor_name* was opened. If the delete statement includes the *at connection_name* clause, the clause must match the *at connection_name* clause of the open cursor statement that opened *cursor_name*.
- The delete statement fails if the cursor was declared for read only, or if the select statement included an order by clause.

See also

close, declare cursor, fetch, open, update

delete (searched)

Description	Removes rows specified by search conditions.
Syntax	exec sql [at <i>connection_name</i>] delete <i>table_name_1</i> [from <i>table_name_n</i>

```
[, table_name_n]...]  
[where search_conditions];
```

Parameters

table_name_1

The name of the table from which this delete statement deletes rows.

from *table_name_n*

The name of a table to be joined with *table_name_1* to determine which rows of *table_name_1* will be deleted. The delete statement does *not* delete rows from *table_name_n*.

where *search_conditions*

Specifies which rows will be deleted. If you omit the where clause, the delete statement deletes all rows of *table_name_1*.

Examples

```
/*  
** Function to FAKE a cascade delete of an author **  
**by name -- this function assumes that pubs2 is  
** the current database.  
** Returns 1 for success, 0 for failure  
**/  
int          drop_author(fname, lname)  
char         *fname;  
char         *lname;  
{  
exec sql begin declare section;  
    CS_CHAR          f_name[41], l_name[41];  
    CS_CHAR          titleid[10], auid[10];  
exec sql end declare section;  
    long            SQLCODE;  
strcpy(f_name, fname);  
strcpy(l_name, lname);  
exec sql whenever sqlerror goto roll_back;  
exec sql select au_id from authors into :auid  
    where au_fname = :f_name  
    and au_lname = :l_name;  
exec sql delete from au_pix where au_id = :auid;  
exec sql delete from blurbs where au_id = :auid;  
exec sql declare curl cursor for  
    select title_id from titleauthor  
    where au_id = :auid;  
exec sql open curl;  
while (SQLCODE == 0)  
{  
    exec sql fetch curl into :titleid;  
    if(SQLCODE == 100) break;  
    exec sql delete from salesdetail  
        where title_id = :titleid;
```

```

exec sql delete from rowsched
    where title_id = :titleid;
exec sql delete from titles
    where title_id = :titleid;
exec sql delete from titleauthor
    where current of curl;
}
exec sql close curl;
exec sql delete from authors
    where au_id = :auid;
exec sql commit work;
return 1;

roll_back:
    exec sql rollback work;
    return 0;
}

```

Usage	<ul style="list-style-type: none"> • This reference page describes mainly aspects of the Transact-SQL delete statement that differ when used with Embedded SQL. See the <i>Adaptive Server Enterprise Reference Manual</i>. • If you need to remove rows specified by the current position of a cursor pointer, use the delete (positioned cursor) statement.
See also	close, declare cursor, fetch, open, update

describe input (SQL descriptor)

Description	<p>Obtains information about dynamic parameter markers in a prepared dynamic SQL statement and stores that information in a SQL descriptor.</p> <p>For a list of possible SQL descriptor datatype codes, see Table 10-5 on page 182.</p>
Syntax	<pre>exec sql describe input <i>statement_name</i> using sql descriptor <i>descriptor_name</i>;</pre>
Parameters	<p><i>statement_name</i></p> <p>The name of the prepared statement about which you want information. <i>statement_name</i> must identify a prepared statement.</p> <p>sql descriptor</p> <p>Identifies <i>descriptor_name</i> as a SQL descriptor.</p>

descriptor_name

The name of the SQL descriptor that can store information about the dynamic parameter markers in the prepared statement.

Examples

```
exec sql begin declare section;
char          query[maxstmt];
int           nin, nout, i;
exec sql end declare section;
int          j;

...

exec sql allocate descriptor din with max 256;
exec sql allocate descriptor dout with max 256;
exec sql whenever sqlerror stop;
exec sql prepare dynstmt from :query;
exec sql describe input dynstmt
      using sql descriptor din;
exec sql get descriptor din :nin = count;
for (i = 0; i < nin; i++)
```

Usage

- Information about the statement is written into the descriptor provided in the using clause. Use the get descriptor statement after executing the describe input statement to extract information from the descriptor into host variables.
- The descriptor must be allocated before the describe input statement can be executed.

See also

allocate descriptor, deallocate descriptor, describe output, get descriptor, prepare, set descriptor

describe input (SQLDA)

Description

Obtains information about dynamic parameter markers in a prepared dynamic SQL statement and stores that information in a SQLDA structure.

Syntax

```
exec sql describe input statement_name
      using descriptor descriptor_name;
```

Parameters

statement_name

The name of the prepared statement about which you want information. *statement_name* must identify a prepared statement.

descriptor

Identifies *descriptor_name* as an SQLDA structure.

descriptor_name

The name of the SQLDA structure that can store information about the dynamic parameter markers in the prepared statement.

Examples

```
...
exec sql prepare s4 from :str4;
exec sql declare c2 cursor for s4;
exec sql describe input s4 using descriptor dinout;
printf("Number of input parameters is %hd\n",
      dinout.sd.sqlld);
```

Usage

- Information about the statement is written into the descriptor specified in the using clause. After the get descriptor statement is executed, you can read the information out of the SQLDA structure.

See also

allocate descriptor, deallocate descriptor, describe output, get descriptor, prepare, set descriptor

describe output (SQL descriptor)

Description

Obtains row format information about the result set of a prepared dynamic SQL statement.

For a list of possible SQL descriptor datatype codes, see Table 10-5 on page 182.

Syntax

```
exec sql describe [output] statement_name
using sql descriptor descriptor_name;
```

Parameters

output

An optional keyword that has no effect on the describe output statement but provides conformance to the SQL standard.

statement_name

The name (specified in a prepare statement) that represents the select statement to be executed.

sql descriptor

Identifies *descriptor_name* as a SQL descriptor.

descriptor_name

The name of a SQL descriptor that is to store the information returned by the describe output statement.

Examples

```
...
exec sql open curs2 using sql descriptor descr_out;
```

```

exec sql describe output prep_stmt4
      using sql descriptor descr_out;
while (sqlca.sqlcode != 100 && sqlca.sqlcode >= 0)
{
      exec sql fetch curs2 into sql descriptor
            descr_out;
      print_descriptor();
}
exec sql close curs2;
exec sql deallocate descriptor descr_out;
exec sql deallocate prepare prep_stmt4;
printf("dynamic sql method 4 completed\n\n");
}
...

```

- Usage
- The information obtained is the type, name, length (or precision and scale, if a number), nullable status, and number of items in the result set.
 - The information is about the result columns from the select column list.
 - Execute this statement before the prepared statement executes. If you perform a describe output statement after you execute and before you perform a get descriptor, the results will be discarded.

See also allocate descriptor, describe input, execute, get descriptor, prepare

describe output (SQLDA)

Description	Obtains row format information about the result set of a prepared dynamic SQL statement and stores that information in a SQLDA structure.
Syntax	<code>exec sql describe [output] <i>statement_name</i> using descriptor <i>sqlda_name</i>;</code>
Parameters	<p><code>output</code> An optional keyword that has no effect on the describe output statement but provides conformance to the SQL standard.</p> <p><code>statement_name</code> The name (specified in a prepare statement) that represents the select statement to be executed.</p> <p><code>descriptor</code> Identifies <i>sqlda_name</i> as a SQLDA structure.</p>

sqlda_name

The name of a SQLDA structure that will store the information returned by the describe output statement.

Examples

```
...
exec sql open curs2 using descriptor input_descriptor;
exec sql describe output statement using descriptor
    output_descriptor;
output_descriptor->sqlda_column->sqlda_sqldata = character;
output_descriptor->sqlda_column->sqlda_datafmt.datatype =      CS_CHAR_TYPE;
output_descriptor->sqlda_column->sqlda_datafmt.maxlength = 20;
output_descriptor->sqlda_column->sqlda_sqllen = 20;
output_descriptor->sqlda_column->sqlda_datafmt.format =
    (CS_FMT_NULLTERM | CS_FMT_PADBLANK);
exec sql fetch curs2 into descriptor output_descriptor;
```

Usage

- The information obtained is the data held in the SQLDA fields, such as the type, name, length (or precision and scale, if a number), nullable status, and number of items in the result set.
- The information is about the result columns from the select column list.

See also

describe input, execute, prepare

disconnect

Description

Closes one or more connections to a Adaptive Server Enterprise.

Syntax

```
exec sql disconnect
{connection_name | current | DEFAULT | all};
```

Parameters

connection_name

The name of a connection to be closed.

current

Specifies that the current connection will be closed.

DEFAULT

Specifies that the default connection is to be closed. This keyword must be in uppercase letters if you specify the default *connection_name* using a character string variable, for example:

```
exec sql disconnect :hv;
```

all
Specifies that all active connections be closed.

Examples

```
#include <stdio.h>

exec sql include sqlca;

main()
{
    exec sql begin declare section;
    CS_CHAR servname[31], username[31],
    password[31], conname[129];
    exec sql end declare section;

    exec sql whenever sqlerror call error_handler();
    exec sql whenever sqlwarning call error_handler();
    exec sql whenever not found continue;

    printf ("Username: ");
    gets  (username);
    printf ("Password: ");
    gets  (password);
    printf ("Adaptive Server Enterprise name: ");
    gets  (servname);
    printf ("Connection name: ");
    gets  (conname);

    /*
    ** Make a named connection.
    */
        exec sql connect :username identified by :password
            at :conname using :servname;
    /*
    ** Make an unnamed (default) connection.
    */
        exec sql connect :username identified by :password
            using :servname;
    /*
    ** The second (default) connection is the current connection.
    */
        exec sql disconnect current;
    /*
    ** We now have neither a default connection nor a current one.
    */
        exec sql disconnect :conname;
```

```

/*
** Now there are no open connections.
*/
    exec sql exit;
}

error_handler()
{
    printf("%d\n%s\n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
    exit(0);
}

```

- Usage
- By itself, the `disconnect` keyword is not a valid statement. Instead, it must be followed by *connection_name*, `current`, `DEFAULT`, or `all`.
 - Closing a connection releases all memory and resources associated with that connection.
 - `disconnect` does not commit current transactions; it rolls them back. If an unchained transaction is active on the connection, `disconnect` rolls it back, ignoring any savepoints.
 - Closing a connection closes open cursors, drops temporary Adaptive Server Enterprise objects, releases any locks the connection has in the Adaptive Server Enterprise, and closes the network connection to the Adaptive Server Enterprise.

See also `commit work`, `commit transaction`, `connect`, `rollback transaction`, `rollback work`

EXEC

Description Runs a system procedure or a user-defined stored procedure.

Syntax

```

exec sql [at connection_name]
    exec [[:status_var =]status_value] procedure_name
    [([@parameter_name =]param_value [out[put]]],...)
    [into :hostvar_1 [:indicator_1]
    [, hostvar_n [indicator_n,...]]]
    [with recompile];

```

Note Do not confuse the `exec` statement with the Embedded SQL `execute` statement; they are not related. The Embedded SQL `exec` statement is, however, the equivalent of the Transact-SQL `execute` statement.

Parameters

status_var

A host variable to receive the return status of the stored procedure.

status_value

The value of the stored procedure return status variable *status_var*.

procedure_name

The name of the stored procedure to be executed.

parameter_name

The name(s) of the stored procedure's parameter(s).

param_value

A host variable or literal value.

output

Indicates that the stored procedure returns a parameter value. The matching parameter in the stored procedure must also have been created using the output keyword.

into *:hostvar_1*

Causes row data returned from the stored procedure to be stored in the specified host variables (*hostvar_1* through *hostvar_n*). Each host variable can have an indicator variable.

with recompile

Causes Adaptive Server Enterprise to create a new query plan for this stored procedure each time the procedure executes.

Examples

Example 1

```
exec sql begin declare section;
      char      titleid[10];
      int       total_discounts;
      short     retstat;
exec sql end declare section exec;
exec sql create procedure get_sum_discounts
      (@titleid tid, @discount int output) as
begin
      select @discount = sum( qty * discount)
      from salesdetail
      where title_id = @titleid

end;
printf("title id: ");
gets(titleid);

exec sql exec
      :retstat = get_sum_discount :titleid,
```

```

        :total_discounts out;

printf("total discounts for title_id %s were
      %s\n", titleid, total_discounts);
exec sql begin declare section;
      CS_INT          status;
      CS_CHAR         city(30);
      CS_INT          result;
exec sql end declare section;
LONG                SQLCODE;

input "City", city ;
exec sql exec countcity :city, :result out;
if (SQLCODE = 0)
    print city + " occurs " + result + "
      times." ;

```

Example 2

```

EXEC SQL BEGIN DECLARE SECTION;
/* storage for login name and password */
CS_CHAR          username[30], password[30];
CS_CHAR          pub_id[4][5], pub_name[4][40], stmt[100] ;
CS_CHAR          city[4][15], state[4][3];
CS_INT           ret_status;
EXEC SQL END DECLARE SECTION ;
...
EXEC SQL set chained off;
strcpy(stmt,"create proc get_publishers as select * from publishers
return ");
EXEC SQL EXECUTE IMMEDIATE :stmt;

EXEC SQL EXEC :ret_status = get_publishers INTO
      :pub_id,
      :pub_name,
      :city,
      :state;

printf("Pub Id      Publisher Name          City          State \n");
printf("\n----- -----\n");
for ( i = 0 ; i < sqlca.sqlerrd[2] ; i++ )
{
    printf("%-8s", pub_id[i]) ;
    printf("%-25s", pub_name[i]) ;
    printf("%-12s", city[i]) ;
    printf("%-6s\n", state[i]) ;
}
printf("\n(%d rows affected, return status = %d)\n", sqlca.sqlerrd[2],
ret_status);
...

```

}

Usage

- Only one select statement can return rows to the client application.
- If the stored procedure contains select statements that can return row data, you must use one of two methods to store the data. You can either use the into clause of the exec statement or declare a cursor for the procedure. If you use the into clause, the stored procedure must not return more than one row of data, unless the host variables that you specify are arrays.
- The value *param_value* can be a host variable or literal value. If you use the output keyword, *param_value* must be a host variable.
- You can specify the output keyword for *parameter_name* only if that keyword was also used for the corresponding parameter of the create procedure statement that created *procedure_name*.
- The Embedded SQL exec statement works much like the Transact-SQL execute statement.

See also

declare cursor (stored procedure), select

exec sql

Description

Marks the beginning of a SQL statement embedded in a host language program.

Syntax

`exec sql [at connection_name] sql_statement;`

Parameters

at

Causes the SQL statement *sql_statement* to execute at the Adaptive Server connection *connection_name*.

connection_name

The connection name that identifies the Adaptive Server connection where *sql_statement* is to execute. The *connection_name* must be defined as a previous connect statement.

sql_statement

A Transact-SQL statement or other Embedded SQL statement.

Examples

```
exec sql
begin declare section;
char      site1(20);
int      sales1;
exec sql end declare section;
```

```

exec sql connect "user1" identified by "password1"
    using "server1";
exec sql connect "user2" identified by "password2"
    using "server2"
/* Remember that a connection that has not been
   explicitly named has the name of its server */
exec sql at server1 select count(*) from sales
    into :sales1;

site1 = sitename("server1");
exec sql at server2 insert into numsales
    values(:site1, :sales1);

```

Usage

- SQL statements embedded in a host language must begin with `exec sql`. The keywords `exec sql` can appear anywhere that a host language statement can begin.
- The statement *sql_statement* can occupy one or more program lines; however, it must conform to host language rules for line breaks and continuation lines.
- The `at` clause affects only the statement *sql_statement*. The clause does not affect subsequent SQL statements, and does not reset the current connection.
- The `at` clause is not valid when *sql_statement* is one of the following SQL statements:

Table 10-2: Statements that cannot use the `at` clause of `exec sql`

allocate descriptor	begin declare section	connect
deallocate descriptor	declare cursor (dynamic)	end declare section
exit	get diagnostics	include file
include sqlca	set connection	set diagnostics
whenever		

- *connection_name* must be defined in a previous connect statement.
- Each Embedded SQL statement must end with a terminator. In C, the terminator is the semicolon (;).

See also

begin declare section, connect, disconnect, set connection

execute

Description	<p>Executes a dynamic SQL statement from a prepared statement.</p> <p>For details on the execute immediate statement, see “execute immediate” on page 154.</p>
Syntax	<pre>exec sql [at <i>connection_name</i>] execute <i>statement_name</i> [into {<i>host_var_list</i> descriptor <i>descriptor_name</i> sql descriptor <i>descriptor_name</i>}] [using {<i>host_var_list</i> descriptor <i>descriptor_name</i> sql descriptor <i>descriptor_name</i>}];</pre> <hr/> <p>Note Do not confuse the Embedded SQL execute statement with the Embedded SQL exec statement or the Transact-SQL execute statement.</p> <hr/>
Parameters	<p><i>statement_name</i></p> <p>A unique identifier for the statement, defined in a previous prepare statement.</p> <p><i>descriptor_name</i></p> <p>Specifies the area of memory, or the SQLDA structure, that describes the statement’s dynamic parameter markers or select column list.</p> <p>into</p> <p>A clause required when the statement executes a select statement, which must be a single-row select. The target of the into clause can be a SQL descriptor, a SQLDA structure, or a list of one or more Embedded SQL host variables.</p> <p>Each host variable in the <i>host_var_list</i> must first be defined in a declare section. An <i>indicator variable</i> can be associated with a host variable to show when a null data value is retrieved.</p> <p>descriptor</p> <p>Identifies <i>descriptor_name</i> as a SQLDA structure.</p> <p>sql descriptor</p> <p>Identifies <i>descriptor_name</i> as a SQL descriptor.</p>

using

The host variables that are substituted for dynamic parameter markers in *host_var_list*. The host variables, which you must define in a *declare* section, are substituted in the order listed. Use this clause only when *statement_name* contains dynamic parameter markers. The dynamic descriptor can also contain the values for the dynamic parameter markers.

Examples

```
exec sql begin declare section;
    CS_CHAR          dymo_buf(128);
    CS_CHAR          title_id(6);
    CS_INT           qty;
    CS_CHAR          order_no(20);
exec sql end declare section;

dymo_buf = "INSERT salesdetail
(ord_num, title_id, qty) VALUES (:?, :?, :?)"

exec sql prepare ins_com from :dymo_buf;

print "Recording Book Sales";
input "Order number?", order_no;
input "Title ID?", title_id;
input "Quantity sold?", qty;
exec sql execute ins_com
    using :order_no, :title_id, :qty;
exec sql disconnect;
```

Usage

- `execute` is the second step in method 2 of dynamic SQL. The first step is the `prepare` statement.
- `prepare` and `execute` are valid with any SQL statement except a multirow `select` statement. For multirow `select` statements, use either dynamic cursor.
- The statement in *statement_name* can contain dynamic parameter markers (“?”). They mark the positions where host variable values are to be substituted before the statement executes.
- The `execute` keyword distinguishes this statement from `exec`. See “`exec`” on page 147.

See also

`declare section`, `get descriptor`, `prepare`, `set descriptor`

execute immediate

Description	Executes a dynamic SQL statement stored in a character-string host variable or quoted string.
Syntax	<code>exec sql [at <i>connection_name</i>] execute immediate {:<i>host_variable</i> "string"};</code>
Parameters	<p><i>host_variable</i></p> <p>A character-string host variable defined in a declare section. Before calling <code>execute immediate</code>, the host variable should contain a complete and syntactically correct Transact-SQL statement.</p> <p><i>string</i></p> <p>A quoted literal Transact-SQL statement string that can be used in place of <i>host_variable</i>.</p>
Examples	<pre>exec sql begin declare section; CS_CHAR host_var(128); exec sql end declare section; printf("Enter a non-select SQL statement: "); gets(host_var); exec sql execute immediate :host_var;</pre>
Usage	<ul style="list-style-type: none">• Using the <code>execute immediate</code> statement is dynamic SQL Method 1. See Chapter 7, "Using Dynamic SQL" for information about the four dynamic SQL methods.• Except for messages, the statement in <i>host_variable</i> cannot return results to the your program. Thus, the statement cannot be, for example, a <code>select</code> statement.• The Embedded SQL precompiler does not check the syntax of the statement stored in <i>host_variable</i> before sending it to Adaptive Server Enterprise. If the statement's syntax is incorrect, Adaptive Server Enterprise returns an error code and message to your program.• Use <code>prepare</code> and <code>execute</code> (dynamic SQL method 2) to substitute values from host variables into a dynamic SQL statement.• Use <code>prepare</code>, <code>open</code>, and <code>fetch</code> (dynamic SQL method 3) to execute <code>select</code> statements with dynamic SQL statements that return results.
See also	<code>execute</code> , <code>prepare</code>

exit

Description	Closes Client-Library and deallocates all Embedded SQL resources allocated to your program.
Syntax	<code>exec sql exit;</code>
Examples	<pre> exec sql include sqlca; main() { /* The body of the main function goes here, ** including various Embedded SQL statements. */ ... /* The exit statement must be the last ** embedded SQL statement in the program. */ exec sql exit; } /* end of main */ </pre>
Usage	<ul style="list-style-type: none"> • The <code>exit</code> statement closes all connections that your program opened. Also, <code>exit</code> deallocates all Embedded SQL resources and Client-Library resources allocated to your program. • Although the <code>exit</code> statement is valid on all platforms, it is required only on some. See the <i>Open Client and Open Server Programmers Supplement</i>. • You cannot use Client-Library functions after using the <code>exit</code> statement, unless you initialize Client-Library again. See the <i>Open Client Client-Library/C Programmers Guide</i> for information about initializing Client-Library. • The <code>exit</code> statement is a Sybase extension; it is not defined in the SQL standard.
See also	<code>disconnect</code>

fetch

Description	Copies data values from the current cursor row into host variables or a dynamic descriptor.
Syntax	<pre> exec sql [at <i>connection_name</i>] fetch [rebind norebind] <i>cursor_name</i> into {:<i>host_variable</i> [[indicator]:<i>indicator_variable</i>] [:<i>host_variable</i> [[indicator]:<i>indicator_variable</i>]]... </pre>

	<code>descriptor</code> <i>descriptor_name</i> <code>sql descriptor</code> <i>descriptor_name</i>);
Parameters	<code>rebind</code> <code>norebind</code> Specifies whether host variables require rebinding for this fetch statement. The <code>rebind</code> clause overrides precompiler options that control rebinding.
	<i>cursor_name</i> The name of the cursor. The name is defined in a preceding <code>declare cursor</code> statement.
	<i>host_variable</i> A host language variable defined in a <code>declare</code> section.
	<i>indicator_variable</i> A 2-byte host variable declared in a previous <code>declare</code> section. If the value for the associated variable is null, <code>fetch</code> sets the indicator variable to -1. If truncation occurs, <code>fetch</code> sets the indicator variable to the actual length of the result column. Otherwise, it sets the indicator variable to 0.
	<code>descriptor</code> Identifies <i>descriptor_name</i> as a SQLDA structure.
	<code>sql descriptor</code> Identifies <i>descriptor_name</i> as a SQL descriptor.
	<i>descriptor_name</i> The name of the dynamic descriptor that will hold the result set.

Examples

```
exec sql begin declare section;
      CS_CHAR          title_id[6];
      CS_CHAR          title[80];
      CS_CHAR          type[12];
      CS_SMALLINT     i_title;
      CS_SMALLINT     i_type;
exec sql end declare section;
exec sql declare title_list cursor for
      select type, title_id, title from titles
      order by type;

exec sql open title_list
while (sqlca.sqlcode != 100) {
exec sql fetch title_list into
      :type :i_type, :title_id, :title :i_title;

      if (i_type != -1) {
          printf("Type: %s\n", type);
      }
      else {
```

```

        printf("Type: undecided\n");
    }

    printf("Title id: %s\n", title_id);

    if (i_title <> -1) {
        print "Title: ", title;
    }
    else {
        print "Title: undecided";
    }
}

exec sql close title_list;

```

Usage

- The fetch statement can be used both with static cursors and with cursors in dynamic SQL.
- The open statement must execute before the fetch statement executes.
- The first fetch on an open cursor returns the first row or group of rows from the cursor's result table. Each subsequent fetch returns the next row or group of rows.
- You can fetch multiple rows into an array.
- The “current row” is the row most recently fetched. To update or delete it, use the where current of *cursor_name* clause with the update or delete statement. These statements are not valid until after a row has been fetched.
- After all rows have been fetched from the cursor, calling fetch sets SQLCODE to 100. If the select furnishes no results on execution, SQLCODE is set to 100 on the first fetch.
- There must be one—and only one—*host_variable* for each column of the result set.
- When neither the rebind nor the norebind is specified, the binding behavior is determined by the precompiler option -b. See “Guidelines for using persistent binding” on page 113 for information on persistent binds and the *Open Client and Open Server Programmers Supplement* for your platform for details on precompiler options.
- An *indicator_variable* must be provided for a *host_variable* that can receive a null value. A runtime error occurs when a null value is fetched for a host variable that has no indicator variable.

- When possible, Client-Library converts the datatype of a result column to the datatype of the corresponding host variable. If Client-Library cannot convert a datatype, it issues an error message. If conversion is not possible, an error occurs.

See also allocate descriptor, close, declare, delete (positioned cursor), open, prepare, update

fetch scrollable cursor

Description	Uses a fetch statement to retrieve data through a cursor and assign it to host variables.
Syntax	<code>exec sql [at connect_name] fetch [fetch orientation]cursor_name into :host_variable [[indicator]: indicator_variable] [,: host_variable [[indicator]: indicator_variable]...];</code>
Parameters	<p><i>host_variable</i> One <i>host_variable</i> exists for each column in the result rows.</p> <p>fetch orientation Specifies the fetch direction of the row to be fetched, if a cursor is scrollable.</p>
Examples	<pre> /* ** Fetch the first row in cursor resultset */ EXEC SQL FETCH FIRST FROM typelist INTO :a_type; printf("\n%s\n", a_type); /* ** Fetch the last row in cursor resultset */ EXEC SQL FETCH LAST FROM typelist INTO :a_type; printf("\n%s\n", a_type); </pre>
Usage	<ul style="list-style-type: none"> • When using <i>host_variable</i>, prefix each host variable with a colon, and separate it from the next host variable with a comma. The host variables listed in the fetch statement must correspond to Adaptive Server Enterprise values that the select statement retrieves. Thus, the number of variables must match the number of returned values, they must be in the same order, and they must have compatible datatypes.

- The options for *fetch orientation* are: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *fetch_offset* and RELATIVE *fetch_offset*. If *fetch orientation* is not specified, next is default. If *fetch orientation* is specified, the cursor must be scrollable. The data that the fetch statement retrieves depends on the cursor position.
The fetch statement typically retrieves single or multiple rows from the cursor result set, depending on the ROW_COUNT specification at cursor open time. If a cursor is not scrollable, fetch retrieves the next row in the result set. If a cursor is scrollable, commands in the fetch statement specify the row position to be fetched.

See also `declare scrollable cursor`

get descriptor

Description	Retrieves attribute information about dynamic parameter markers and select column list attributes and data from a SQL descriptor. For a list of SQL descriptor datatype codes, see Table 10-5 on page 182.
Syntax	<pre>exec sql get descriptor <i>descriptor_name</i> {:<i>host_variable</i> = count value <i>item_number</i> :<i>host_variable</i> = <i>item_name</i> [, :<i>host_variable</i> = <i>item_name</i>]...};</pre>
Parameters	<p><i>descriptor_name</i> The name of the SQL descriptor that contains information about the dynamic parameter markers or return columns in a prepared statement.</p> <p><i>host_variable</i> A variable defined in a declare section.</p> <p>count The number of dynamic parameters retrieved.</p> <p><i>item_number</i> A number specifying the <i>n</i>th dynamic parameter marker or select column for which get descriptor retrieves information.</p> <p><i>item_name</i> The name of an attribute to be retrieved. See Table 10-3:</p>

Table 10-3: Valid *item_name* values

Value	Description
<i>data</i>	Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.
<i>indicator</i>	Value for the indicator parameter associated with the dynamic parameter marker or target.
<i>length</i>	The length, in characters, of the dynamic parameter marker or target for the specified SQL descriptor.
<i>name</i>	The name of the specified SQL descriptor containing information about the dynamic parameter markers.
<i>nullable</i>	Equals 0 if the dynamic parameter marker can accept a null value; otherwise, equals 1.
<i>precision</i>	An integer specifying the total number of digits of precision for the CS_NUMERIC variable.
<i>returned_length</i>	The length of character types of the values from the select column list.
<i>scale</i>	An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.
<i>type</i>	The datatype of this column (item number) in the row. For values, see SQL descriptor datatype codes.

Examples

```
exec sql begin declare section;
    int    numcols, colnum, type, intbuf;
    char   charbuf[100];
exec sql end declare section;
...
exec sql allocate descriptor big_desc
    with max 1000;
exec sql prepare dynstmt from "select * from \
    huge_table";
exec sql execute dynstmt into sql descriptor
    big_desc;
exec sql get descriptor big_desc :numcols = count;
for (colnum = 1; colnum <= numcols; colnum++)
{
    exec sql get descriptor big_desc
        value :colnum :type = type;
    if (type == 4)
    {
```



```

exec sql get descriptor big_desc
  value :colnum :intbuf = data;
/* Display intbuf. */
...
}
else if (type == 1)
{
  big_desc
  value :colnum :charbuf = data;
/* Display charbuf. */
...
}
}
exec sql deallocate descriptor big_desc;
...

```

Usage	<ul style="list-style-type: none"> • The <code>get descriptor</code> statement returns information about the number or attributes of dynamic parameters specified or the select list columns in a prepared statement. • This statement should be executed after a <code>describe input</code>, <code>describe output</code>, <code>execute</code>, or <code>fetch (dynamic)</code> statement has been issued. • It is not possible to retrieve <i>data</i>, <i>indicator</i>, or <i>returned_length</i> until the data associated with the descriptor is retrieved from the server by an <code>execute</code> statement or a <code>fetch</code> statement.
See also	<code>describe input</code> , <code>describe output</code> , <code>fetch</code> , <code>set descriptor</code>

get diagnostics

Description	Retrieves error, warning, and informational messages from Client-Library.
Syntax	<pre> get diagnostics {:hv = statement_info [, :hv = statement_info]... exception :condition_number :hv = condition_info [, :hv = condition_info]...} </pre>
Parameters	<p><i>statement_info</i></p> <p>The keyword <code>number</code> is currently the only supported <i>statement_info</i> type. It returns the total number of exceptions in the diagnostics queue.</p>

condition_info

Any one of the keywords *sqlca_info*, *sqlcode_number*, and *returned_sqlstate*.

Examples

```
exec sql begin declare section;
      CS_INT   num_msgs;
      CS_INT   condcnt=1;
      exec sql include sqlca;
exec sql end declare section;
exec sql exec sp_password "bass", "foo";
exec sql get diagnostics :num_msgs = number;

printf("Number of messages is %d.\n", num_msgs);

/* Loop through and print the messages. */

while (condcnt <= num_msgs)
{
      exec sql get diagnostics exception :condcnt
      :sqlca = sqlca_info;
      printf("SQLCODE = %d \n", sqlca.sqlcode);
      printf("%s \n", sqlca.sqlerrm.sqlerrmc);
      condcnt = condcnt + 1;
}
```

Usage

- Many Embedded SQL statements are capable of causing multiple warnings or errors. Typically, only the first error is reported using SQLCODE, SQLCA, or SQLSTATE. Use get diagnostics to process all the errors.
- You can use get diagnostics, which is the target of the call, perform, or go to clause of a whenever statement, in the code.
- You can use get diagnostics after a statement for which you want to retrieve informational messages.

See also

whenever

include "filename"

Description

Includes an external file in an Embedded SQL source file.

Syntax

exec sql include "filename";

Parameters

filename

The name of the file to be included in the Embedded SQL source file containing this statement.

Examples

```

common.h:
    /* This file contains definitions and
       ** declarations used in the file getinfo.c.
       */

#include <stdio.h>
#include "./common.h"
void    err_handler();
void    warning_handler();
exec sql include sqlca;
{
    exec sql begin declare section;
           CS_CHAR username[33], password[33], date[33];
    exec sql end declare section;

    exec sql whenever sqlerror call err_handler();
    exec sql whenever sqlwarning call warning_handler();
    exec sql whenever not found continue;

/*
** Copy the user name and password defined in common.h to
** the variables decalred for them in the declare section.
*/
strcpy (username, USER);
strcpy(password, PASSWORD);

printf("Today's date: %s\n", date);
...
}
void    err_handler()
{
...
}
void    warning_handler()
{
...
}
/* common.h */
#define USER "sa"
#define PASSWORD ""
=====

```

```
exec sql begin declare section;
    char    global_username[100];
    char    global_password[100];
exec sql end declare section;
```

getinfo.c

```
#include <common.h>
printf("uid?\n");
gets(global_username);
printf("password?\n");
gets(global_password);
```

do_connect.c

```
exec sql include "common.h";

exec sql connect :global_username
    identified by :global_password;
```

Usage

- The Embedded SQL precompiler processes the included file as though it were part of the Embedded SQL source file, recognizing all declare sections and SQL statements. The Embedded SQL precompiler writes the resulting host language source code into the generated file.
- Use the include path precompiler command line option to specify the directories to be searched for any included files. Refer to the *Open Client and Open Server Programmers Supplement*.
- Included files can be nested up to a maximum depth of 32 files.
- The include "filename" statement can be used anywhere.

See also

declare section

include sqlca

Description

Defines the SQL Communications Area (SQLCA) in an Embedded SQL program.

Syntax

```
exec sql include sqlca;
```

Examples

```
exec sql include SQLCA;
...
exec sql update t1 set c1 = 123 where c2 > 47;
if (sqlca.sqlcode == 0)
```

```

    {
        printf("%d rows updated/n", sqlca.sqlerrd[2]);
    }
    else if (sqlca.sqlcode == 100)
    {
        printf("No rows matched the query\n");
    } else {
        printf("An error ocured\n%s\n",
            sqlca.sqlerrm.sqlerrmc);
    }
}

```

- Usage
- The include `sqlca` statement can be used anywhere that host language declarations are allowed.
- See also
- begin declare section

include sqlda

Description Defines the `SQLDA` structure in an Embedded SQL program.

Syntax `exec sql include sqlda;`

Examples

```

exec sql include sqlda;
...
SQLDA *input_descriptor, *output_descriptor;
CS_SMALLINT small;
CS_CHAR character[20];
input_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
input_descriptor->sqlda_sqln = 3;
output_descriptor = (SQLDA *)malloc(SYB_SQLDA_SIZE(3));
output_descriptor->sqlda_sqln = 3;

```

- Usage
- The include `sqlda` statement can be used anywhere that host language declarations are allowed.

initialize_application

Description Generates a call to set the application name on the global `CS_CONTEXT` handle. If precompiled with the `-x` option, it will also set the `cs_config(CS_SET, CS_EXTERNAL_CONFIG, CS_TRUE)` property.

Syntax

```
exec sql initialize_application
[application_name "=" application_name];
```

Examples

```
exec sql include sqlca;
main()
{
exec sql initialize_application
application_name = :appname;
/*
** The body of the main function goes here,
** including various Embedded SQL statements.
*/
...
/* The init statement must be the first
** embedded SQL statement in the program.
*/
exec sql exit;
} /* end of main */
```

Usage

- *application_name* is either a literal string or a character variable containing the name of the application.
- If *initialize_application* is the *first* Embedded SQL statement executed by an application, *-x* causes *ct_init* to use external configuration options to initialize the Client-Library part of the CS_CONTEXT structure.
- If *initialize_application* is not the first Embedded SQL statement, *ct_init* does *not* pick up external configuration options.
- Regardless of whether or not *initialize_application* is the first Embedded SQL statement, *-x* causes *exec sql connect* statements to use external configuration data. If *-e* is also specified, Sybase uses the server name as a key to the configuration data. If *-e* is not specified, then the application name (or DEFAULT) is used as the key to the configuration data.
- If you specify *-x* and the application name, the following applies:
 - *ct_init* uses the application name to determine which section of the external configuration file to use for initialization.
 - The application name is passed to Adaptive Server Enterprise as part of the connect statement. The application name is entered in the *sysprocesses.program_name* table.
- If *-e* is specified without *-x*, then *ct_init* will use external configuration data when initializing, but every connection will use the server name as a key to the external configuration data. See the *Open Client and Open Server Programmers Supplement* for information on command-line options.

See also `exit`

open (dynamic cursor)

Description	Opens a previously declared dynamic cursor.
Syntax	<code>exec sql [at <i>connection_name</i>] open <i>cursor_name</i> [row_count = <i>size</i>] [using {<i>host_var_list</i> descriptor <i>descriptor_name</i> sql descriptor <i>descriptor_name</i>}]</code> ;
Parameters	<p><i>cursor_name</i></p> <p>Names a cursor that has been declared using the declare cursor statement.</p> <p><i>size</i></p> <p>The number of rows moved in a network roundtrip, not the number fetched into the host variable. The <i>size</i> argument can be either a literal or a declared host variable.</p> <p><i>host_var_list</i></p> <p>Names the host variables that contain the values for dynamic parameter markers.</p> <p>descriptor</p> <p>Identifies <i>descriptor_name</i> as a SQLDA structure.</p> <p>sql descriptor</p> <p>Identifies <i>descriptor_name</i> as a SQL descriptor.</p> <p><i>descriptor_name</i></p> <p>Names the dynamic descriptor that contains information about the dynamic parameter markers in a prepared statement.</p>
Examples	<pre>exec sql begin declare section; CS_CHAR dyna_buf[128]; CS_CHAR title_id[6]; CS_CHAR lastname[40]; CS_CHAR firstname[20]; CS_CHAR phone[12]; exec sql end declare section; dyna_buf = "SELECT a.au_lname, a.au_fname, a.phone" + "FROM authors a, titleauthor t " + "WHERE a.au_id = t.au_id " + "AND t.title_id = ?";</pre>

```
exec sql prepare dyna_comm from :dyna_buf;

exec sql declare who_wrote cursor for dyna_comm;

printf("List authors for what title? ");
gets(title_id);
exec sql open who_wrote using :title_id;
while (TRUE){          exec sql fetch who_wrote into
                      :lastname, :firstname, :phone;
                      if (sqlcode == 100) break;
                      printf("Last name is %s\n",lastname,
"First name is %s\n", firstname,
                      "Phone number is %s\n", phone);
                      }

exec sql close who_wrote;
```

Usage

- open executes the statement specified in the corresponding declare cursor statement. You can then use the fetch statement to retrieve the results of the prepared statement.
- You can have any number of open cursors.
- The using clause substitutes host-variable or dynamic-descriptor contents for the dynamic parameter markers (“?”) in the select statement.

See also

close, declare, fetch, prepare

open (static cursor)

Description

Opens a previously declared static cursor. This statement can be used to open any static cursor, including one for a stored procedure.

Syntax

```
exec sql [at connection_name] open cursor_name
[row_count = size];
```

Parameters

cursor_name

The name of the cursor to be opened.

row_count

The number of rows moved in a network roundtrip, not the number fetched into the host variable.

size

The number of rows that are moved at the same time from Adaptive Server Enterprise to the client. The client buffers the rows until they are fetched by the application. This parameter allows you to tune network efficiency.

Examples

```

exec sql begin declare section;
    char          b_titleid[tidsize+1];
    char          b_title[65];
    char          b_type[typesize+1];
exec sql end declare section;
    long          sqlcode;
    char          response[10];

    ...

exec sql declare titlelist cursor for
    select title_id, substring(title,1,64)
    from titles where type like :b_type;
strcpy(b_type, "business");
exec sql open titlelist;
for (;;)
    exec sql fetch titlelist into :b_titleid,
        :b_title;
    if (sqlcode == 100)
        break;
    printf("  %-8s %s\n", b_titleid, b_title);
    printf("update/delete? ");
    gets(response);
    if (!strncasecmp(response,"u",1))
    {
        printf("enter the new titleid\n>");
        gets(b_titleid);
        exec sql update titles
            set title_id = :b_titleid
            where current of titlelist;
    }
    else if (!strncasecmp(response,"d",1))
    {
        exec sql delete from titles
            where current of titlelist;
    }
}
exec sql close titlelist;

```

Usage

- open executes the select statement given by the declare cursor statement and prepares results for the fetch statement.
- You can have an unlimited number of open cursors.

- A static cursor must be opened only in the file where the cursor is declared. The cursor can be closed in any file.
- The values of host variables embedded in the declare cursor statement are taken at open time.
- When specifying *cursor_name*, you can use the name of a deallocated static cursor. If you do, the precompiler declares and opens a new cursor having the same name as that of the deallocated cursor. Thus, the precompiler does not reopen the deallocated cursor but instead creates a new one. The results sets for the two cursors can differ.

prepare

Description	Declares a name for a dynamic SQL statement buffer.
Syntax	<code>exec sql [at <i>connection_name</i>] prepare <i>statement_name</i> from {:<i>host_variable</i> "<i>string</i>";</code>
Parameters	<p><i>statement_name</i></p> <p>An identifier used to reference the statement. <i>statement_name</i> must uniquely identify the statement buffer and must conform to the SQL identifier rules for naming variables. The <i>statement_name</i> can also be a <i>host_variable</i> string containing a valid SQL identifier. <i>statement_name</i> can be up to 255 characters.</p> <p><i>host_variable</i></p> <p>A character-string host variable that contains an executable SQL statement. Place dynamic parameter markers (“?”) anywhere in the select statement where a host variable value will be substituted.</p> <p><i>string</i></p> <p>A literal string that can be used in place of <i>host_variable</i>.</p>
Examples	<pre>exec sql begin declare section; CS_CHAR dyn_buffer [128]; CS_CHAR state [2]; exec sql end declare section; -- The select into table_name statement returns no -- results to the program, so it does not -- need a cursor. dyn_buffer = "select * into #work from authors"</pre>

```

+ "where state = ?";

printf("State? ");
gets(state);
exec sql prepare make_work from :dyn_buffer;
exec sql execute make_work using :state;

```

Usage

- In the current implementation, Sybase creates a temporary stored procedure for a dynamic SQL statement stored in a character string literal or host variable.
- `prepare` sends the contents of *host_variable* to the Adaptive Server Enterprise to convert into a temporary stored procedure. This temporary stored procedure remains in `tempdb` on Adaptive Server Enterprise until the statement is deallocated or the connection is disconnected.
- The scope of *statement_name* is global to your program but local to the connection *connection_name*. The statement persists until the program either deallocates it or closes the connection.
- `prepare` is valid with Dynamic SQL methods 2, 3, and 4.
- With method 2, (`prepare` and `execute`), an `execute` statement substitutes values from host variables, if any, into the prepared statement and sends the completed statement to Adaptive Server Enterprise. If there are no host variables to substitute and no results, you can use `execute immediate`, instead.
- With method 3, `prepare` and `fetch`, a `declare cursor` statement associates the saved `select` statement with a cursor. An open statement substitutes values from host variables, if any, into the `select` statement and sends the result to Adaptive Server Enterprise for execution.
- With methods 2, 3, and 4, `prepare` and `fetch` with parameter descriptors, the dynamic parameter descriptors, represented by question marks (“?”), indicate where host variables will be substituted.
- A prepared statement must be executed on the same connection on which it was prepared. If the prepared statement is used to declare a cursor, all operations on that cursor use the same connection as the prepared statement.
- The statement in *host_variable* can contain dynamic parameter markers that indicate where to substitute values of host variables into the statement.

See also

`declare cursor`, `execute`, `execute immediate`, `deallocate prepare`

rollback

Description	Rolls a transaction back to a savepoint inside the transaction or to the beginning of the transaction.
Syntax	<pre>exec sql [at <i>connection_name</i>] rollback [transaction tran work] [<i>transaction_name</i> <i>savepoint_name</i>];</pre>
Parameters	<p>transaction trans work</p> <p>The keywords transaction, trans, and work are interchangeable in the rollback statement, but only work is ANSI-compliant.</p> <p><i>transaction_name</i></p> <p>The name of the transaction being rolled back.</p> <p><i>savepoint_name</i></p> <p>The name assigned to the savepoint in a save transaction statement. If you omit <i>savepoint_name</i>, Adaptive Server rolls back the entire transaction.</p>
Examples	<pre>abort_tran: exec sql whenever sqlerror continue: exec sql at connect2 rollback transaction; exec sql at connect1 rollback transaction; goto try_update;</pre>
Usage	<ul style="list-style-type: none">• This reference page mainly describes aspects of the Transact-SQL rollback statement that differ when used with Embedded SQL. See the <i>Adaptive Server Enterprise Reference Manual</i>.• Transaction names and savepoint names must conform to the Transact-SQL rules for identifiers.• Transaction names and savepoints are Transact-SQL extensions; they are not ANSI-compliant. Do not use a transaction name or savepoint name with the ANSI-compliant keyword work.
See also	begin transaction, commit

select

Description	Retrieves rows from database objects.
Syntax	<pre>exec sql [at <i>connect_name</i>] select <i>select_list</i></pre>

into *destination*
from *table_name...*;

Parameters

select_list

Same as *select_list* in the Transact-SQL select statement, except that the *select_list* cannot perform variable assignments in Embedded SQL.

destination

A table or a series of one or more Embedded SQL host variables. Each host variable must first be defined in a previous declare section. *Indicator variables* can be associated with the host variables.

Examples

```

/* This example retrieves columns from a
** single row of the authors table and
** stores them in host variables. Because the
** example's select statement cannot return more
** than one row, no cursor is needed.
*/

exec sql begin declare section;
    character        last[40];
    character        first[20];
    character        phone[12];
    character        id[11];
exec sql end declare section;

    printf("Enter author id: ");
gets(id);
exec sql select au_lname, au_fname, phone
    into :last, :first, :phone
    from authors
    where au_id = :id;
if (sqlcode != 100)
{
    print "Information for Author ", id, ":";
    print last, first, phone;
}
else
{
    print "Could not locate author ", id;
};

```

Usage

- This reference page mainly describes aspects of the Transact-SQL select statement that differ when the statement is used in Embedded SQL. See the *Adaptive Server Enterprise Reference Manual*.
- The compute clause of the Transact-SQL select statement cannot be used in Embedded SQL programs.

- Host variables in a select statement are input variables only, except in the statement's into clause. Host variables in the into clause are output variables.
- Previously declared input host variables can be used anywhere in a select statement that a literal value or Transact-SQL variable is allowed. Indicator variables can be associated with input host variables to specify null values.
- If a select statement returns more than one row, each host variable in the statement's into clause must be an array with enough space for all the rows. Otherwise, you must use a cursor to bring the rows back one at a time.

See also

declare cursor

set connection

Description Causes the specified existing connection to become the current connection.

Syntax `set connection {connection_name | DEFAULT};`

Parameters *connection_name*

The name of an existing connection that you want to become the current connection.

default

Specifies that the unnamed default connection is to become the current connection.

Examples

```
exec sql connect "ME" at connect1 using "SERVER1";
exec sql connect "ME" at connect2 using "SERVER2";

/* The next statement executes on connect2. */
exec sql select userid() into :myid;

exec sql set connection connect1;

/* The next statement executes on connect1. */
exec sql select count(*) from t1;
```

Usage

- The set connection statement specifies the current connection for all subsequent SQL statements, except those preceded by the exec sql clause at.
- A set connection statement remains in effect until you choose a different current connection by using the set connection statement again.

See also [at connection_name, connect](#)

set descriptor

Description	<p>Inserts or updates data in a SQL descriptor.</p> <p>For a list of possible SQL descriptor datatypes, see Table 10-5 on page 182.</p>
Syntax	<pre>exec sql set descriptor <i>descriptor_name</i> {count = <i>host_variable</i>} {value <i>item_number</i> {<i>item_name</i> = :host_variable}[,...];</pre>
Parameters	<p><i>descriptor_name</i> The name of the SQL descriptor that contains information about the dynamic parameter markers in a prepared statement.</p> <p>count The number of dynamic parameter specifications to be described.</p> <p><i>host_variable</i> A host variable defined in a declare section.</p> <p><i>item_number</i> Represents the <i>n</i>th occurrence of either a dynamic parameter marker or a select column.</p> <p><i>item_name</i> Represents the attribute information of either a dynamic parameter marker or a select list column. Table 10-4 lists the values for <i>item_name</i>.</p>

Table 10-4: Values for item_name

Value	Description
<i>data</i>	Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.
<i>length</i>	The length, in characters, of the dynamic parameter marker or target for the specified SQL descriptor.
<i>precision</i>	An integer specifying the total number of digits of precision for the CS_NUMERIC variable.
<i>scale</i>	An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.
<i>type</i>	The datatype of this column (item number) in the row. For values, see Table 10-5 on page 182.

Examples

```
exec sql prepare get_royalty
      from "select royalty from roysched
      where title_id = ? and lorange <= ? and
      hirange > ?";

exec sql allocate descriptor roy_desc with max 3;
exec sql set descriptor roy_desc
      value 1 data = :tid;
exec sql set descriptor roy_desc
      value 2 data = :sales;
exec sql set descriptor roy_desc
      value 3 data = :sales;
exec sql execute get_royalty into :royalty
      using sql descriptor roy_desc;
```

Usage

An Embedded SQL program passes attribute and value information to Client-Library, which holds the data in the specified SQL descriptor until the program issues it a request to execute a statement.

See also

allocate descriptor, describe input, describe output, execute, fetch, get descriptor, open(dynamic cursor)

thread exit

Description	Allows Embedded SQL programs to release memory allocated to a particular thread.
Syntax	<code>exec sql thread_exit;</code>
Examples	<pre> exec sql include sqlca; main() { ... for (;;) { /* A thread connects to Adaptive Server Enterprise, ** executes various embedded SQL statements, ** and then disconnects from ** Adaptive Server Enterprise */ ... exec sql thread_exit; ... } /* The exit statement must be the last ** embedded SQL statement in the program. */ exec sql exit; } /* end of main */ </pre>
Usage	<ul style="list-style-type: none"> • The thread exit statement deallocates all memory resources allocated to a particular thread. • The thread exit statement is a Sybase extension; it is not defined in the SQL standard.
See also	exit

update

Description	Modifies data in rows of a table.
Syntax	<pre> exec sql [at connection_name] update table_name set [table_name] column_name1 = {expression1 NULL (select_statement)} [, column_name2 = {expression2 NULL </pre>

```
    | (select_statement)]...  
[from table_name  
  [, table_name]...  
[where {search_conditions | current of cursor_name}];
```

Parameters*table_name*

The name of a table or view, specified in any format that is valid for the update statement in Transact-SQL.

Examples

```
exec sql begin declare section;  
    CS_CHAR      store_name[40];  
    CS_CHAR      disc_type[40];  
    CS_INT       lowqty;  
    CS_INT       highqty;  
    CS_FLOAT     discount;  
exec sql end declare section;  
  
CS_CHAR      answer[1]);  
  
exec sql declare update_cursor cursor for  
    select s.stor_name, d.discounttype,  
    d.lowqty, d.highqty, d.discount  
    from   stores s, discounts d  
    where  d.stor_id = s.stor_id;  
  
exec sql open update_cursor;  
  
exec sql whenever not found goto alldone;  
  
while (TRUE) {  
    exec sql fetch update_cursor into  
        :store_name, :disc_type, :lowqty,  
        :highqty, discount;  
    print store_name, disc_type, lowqty,  
        highqty, discount;  
    printf("New discount? ");  
    gets(discount);  
    exec sql update discounts  
        set discount = :discount  
        where current of update_cursor;  
}  
  
alldone:  
exec sql close update_cursor;  
exec sql disconnect all;
```

Usage	<ul style="list-style-type: none"> • This reference page mainly describes aspects of the Transact-SQL update statement that differ when the statement is used in Embedded SQL. See the <i>Adaptive Server Enterprise Reference Manual</i>. • Host variables can appear anywhere in an expression or in any where clause. • You can use the where clause to update selected rows in a table. Omit the where clause to update all rows in the table. Use where current of <i>cursor_name</i> to update the current row of an open cursor. • When where current of <i>cursor_name</i> is specified, the statement must be executed on the connection specified in the open cursor statement. If the at <i>connection_name</i> clause is used, it must match the open cursor statement.
See also	close, delete cursor, fetch, open, prepare

whenever

Description	Specifies an action to occur whenever an executable SQL statement causes a specified condition.
Syntax	<pre>exec sql whenever {sqlerror not found sqlwarning} {continue go to <i>label</i> goto <i>label</i> stop call <i>routine_name</i> [<i>args</i>];</pre>
Parameters	<p>sqlerror Specifies an action to take when an error is detected, such as a syntax error returned to the Embedded SQL program from Adaptive Server.</p> <p>not found Specifies an action to take when a fetch or select into statement retrieves no data or when a searched update or delete statement affects no rows.</p> <p>sqlwarning Specifies an action to take when a warning is received; for example, when a character string is truncated.</p> <p>continue Take no action when the condition occurs.</p> <p>go to goto Transfer control to the program statement at the specified <i>label</i>.</p> <p><i>label</i> A host language statement label, such as a C label.</p>

stop

Terminate the Embedded SQL program when the condition occurs.

call

Transfer control to a callable routine in the program, such as a user-defined function or subroutine.

routine_name

A host language routine that can be called. The routine must be able to be called from the source file that contains the whenever statement. You may need to declare the routine as external to compile the Embedded SQL program.

args

One or more arguments to be passed to the callable routine, using the parameter-passing conventions of the host language. The arguments can be any list of host variables, literals, or expressions that the host language allows. A space character should separate each argument from the next.

Examples

```
exec sql whenever sqlerror call err_handler();
exec sql whenever sqlwarning call warn_handler();
```

```
long SQLCODE;
exec sql begin declare section;
    CS_CHAR      lastname[40];
    CS_CHAR      firstname[20];
    CS_CHAR      phone[12];
exec sql end declare section;

exec sql declare au_list cursor for
    select au_lname, au_fname, phone
    from authors
    order by au_lname;

exec sql open au_list;

exec sql whenever not found go to list_done;

while (TRUE){
    exec sql fetch au_list
        into :lastname, :firstname, :phone;
    printf("Lastname is: %s\n", lastname,
"Firstname is: %s\n", firstname,
"Phone number is: %s\n", phone;
}
list_done:
```

```
exec sql close au_list;  
exec sql disconnect current;
```

Usage

- The `whenever` statement causes the Embedded SQL precompiler to generate code following each executable SQL statement. The generated code includes the test for the condition and the host language statement or statements that carry out the specified action.
- The Embedded SQL precompiler generates code for the SQL statements that follow the `whenever` statement in the source file, including SQL statements in subroutines that are defined in the same source file.
- Use `whenever ...continue` to cancel a previous `whenever` statement. The `continue` action causes the Embedded SQL precompiler to ignore the condition. To prevent infinite loops, use `whenever ...continue` in an error handler before executing any Embedded SQL statements.
- When you use `whenever ...go to label`, `label` must represent a valid location to resume execution. In C, for example, `label` must be declared in any routine that has executable SQL statements within the scope of the `whenever` statement. C does not allow a `goto` statement to jump to a label declared in another function.
- If you have a `whenever` statement in your program but you have not declared `SQLCA` or `SQLSTATE` status variables, the Embedded SQL precompiler assumes that you are using the `SQLCODE` variable. Be sure that `SQLCODE` is declared. Otherwise, the generated code will not compile.

SQL descriptor codes

Table 10-5 pertains to the SQL descriptor used for dynamic SQL statements. Sybase's use of dynamic SQL values conforms to the ANSI/ISO 185-92 SQL-92 standards. See the appropriate ANSI/ISO documentation.

Table 10-5: SQL descriptor datatype codes

ANSI SQL datatype	Code
bit	14
character	1
character varying	12
date, time	9
decimal	3
double precision	8
float	6
integer	4
numeric	2
real	7
smallint	5

Sybase-defined datatype	Client-Library code
smalldatetime	-9
money	-10
smallmoney	-11
text	-3
image	-4
tinyint	-8
binary	-5
varbinary	-6
long binary	-7
longchar	-2

Table 10-6: SQL descriptor identifier values

Value	Description
<i>type</i>	The datatype of this column (item number) in the row. For values, see Table 10-5 on page 182.
<i>length</i>	The length, in characters, of the dynamic parameter marker of target for the specified SQL descriptor.
<i>returned_length</i>	The length of char types of the values from the select column list.
<i>precision</i>	An integer specifying the total number of digits of precision for the CS_NUMERIC variable.

Value	Description
<i>scale</i>	An integer specifying the total number of digits after the decimal point for the CS_NUMERIC variable.
<i>nullable</i>	Equals 0 if the dynamic parameter marker can accept a null value; otherwise, equals 1.
<i>indicator</i>	Value for the indicator parameter associated with the dynamic parameter marker or target.
<i>data</i>	Value for the dynamic parameter marker or target associated with the specified SQL descriptor. If indicator is negative, this field is undefined.
<i>name</i>	The name of the specified SQL descriptor containing information about the dynamic parameter markers.

whenever

Open Client and Open Server Configuration File

Open Client and Open Server applications can easily be configured using the Open Client and Open Server configuration file. By default, the file is named *ocs.cfg* and is located in the `$$SYBASE/$$SYBASE_OCS/config` directory. This chapter describe how the configuration file can be used with Embedded SQL.

Topic	Page
Purpose of the Open Client and Open Server configuration file	185
Accessing the configuration functionality	185
Default settings	186
Syntax for the Open Client and Open Server configuration file	187
Sample programs	189
Summary	195

Purpose of the Open Client and Open Server configuration file

The Open Client and Open Server configuration file provides a single location where all Open Client and Open Server application connections can be configured. Using the configuration file simplifies the tasks of establishing configuration standards and managing configuration changes.

Accessing the configuration functionality

This feature is available through two command-line options of the `initialize_application` statement.

- `-x` – this option allows for external configuration. The application needs to initialize an application with a name. The Open Client and Open Server configuration file will have a section with this application name. Under this section, place all properties that need to be set for this application. The `-x` option is useful only when used with `initialize_application`. If initializing is not done, and the `-x` option is used, the default section of the configuration file will be accessed.
- `-e` – this option allows us to configure by SERVER NAME. No call to `initialize_application` is required. The server name will be used as a key to look up in the configuration file for properties to be set the section defined by the server name. This will allow users to associate connection names with specific connection properties.

Note If `INITIALIZE_APPLICATION` is not the first Embedded SQL statement to be executed, external configuration properties will not be set. If it is the first Embedded SQL statement to be executed, then the external configuration options will be used for initialization.

Default settings

The following is the Open Client and Open Server configuration file with default settings. You can customize the file as needed.

[DEFAULT]

```
;This is the default section loaded by applications that use the
;external configuration feature, but which do not specify their
;own application name. Initially this section is empty.Defaults
;from all properties will be the same as earlier releases of
;Open Client libraries.
```

[ANSI_ESQL]

```
;This section defines configuration which an ANSI conforming
;Embedded SQL application should use to get ANSI-defined
;behavior from Adaptive Server Enterprises and Open Client libraries. This set
of
;configuration ;properties matches the set which earlier
;releases of Embedded SQL (version 10.0.x) automatically set for
;applications duringexecution of a CONNECT statement.
CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
CS_EXTRA_INF=CS_TRUE
CS_ANSI_BINDS=CS_TRUE
```

```

CS_OPT_ANSINULL=CS_TRUE
CS_OPT_ANSIPERM=CS_TRUE
CS_OPT_STR_RTRUNC=CS_TRUE
CS_OPT_ARITHABORT=CS_FALSE
CS_OPT_TRUNCIGNORE=CS_TRUE
CS_OPT_ISOLATION=CS_OPT_LEVEL3
CS_OPT_CHAINXACTS=CS_TRUE
CS_OPT_CURCLOSEONXACT=CS_TRUE
CS_OPT_QUOTED_IDENT=CS_TRUE
;End of default sections

```

Syntax for the Open Client and Open Server configuration file

The syntax for the Open Client and Open Server configuration file will match the existing syntax for Sybase localization and configuration files supported by CS-Library with minor variations.

The syntax is as follows:

- `;` – Signifies a comment line.
- `[section_name]` – Section names are wrapped in square brackets. The Open Client and Open Server configuration file comes with sections named `DEFAULT` and `ANSI_ESQL`. The application name will be used as the section name for an application that has been compiled with the `-x` option. For an application that has been compiled with the `-e` option, the server name will be used for the section name. Any name can be used as a section name for the sections that contain settings that will be used in multiple sections. The following example shows a section arbitrarily named “`GENERIC`,” and how that section is included in other sections:

```

[GENERIC]
  CS_OPT_ANSINULL=CS_TRUE
[APP_PAYROLL]
  include=GENERIC
  CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
[APP_HR]
  include=GENERIC
  CS_OPT_QUOTED_IDENT=CS_TRUE

```

- `entry_name=entry_value`

- Entry values can be anything: integers, strings and so on. If an entry value line ends with `\"<newline>` the entry value continues to the next line.
- White spaces are trimmed from the beginning and end of entry values.
- If white spaces are required at the beginning or end of an entry value, wrap them in double quotes.
- An entry that begins with a double quote must end with a double quote. Two double quote characters in a row within a quoted string represent a single double quote in the value string. If a newline is encountered within double quotes, it is considered to be literally part of the value.
- Entry names and section names can consist of alphabetic characters (both uppercase and lowercase), the digits 0 - 9, and any of the following punctuation characters: `! " # $ % & ' () * + , - . / : ; < > ? @ \ ^ _ ` { | } ~`.

Square brackets ([]), space, and equal sign (=) are not supported. The first letter MUST be alphabetic.

- Entry and section names are case sensitive.
- `Include=earlier_section`

If a section contains the entry `include`, then the entire contents of that previously defined section are considered to be replicated within this section. In other words, the properties defined in the previous section are inherited by this section.

Note that the included section must have been defined prior to it being included in another section. This allows the configuration file parsing to happen in a single pass and eliminates the need to detect recursive included directives.

If an included section in turn includes another section, the order of entry values is defined by a “depthfirst” search of the included sections.

Sections cannot include a reference to themselves. In other words, recursion is not possible because you must include a previously defined section—you cannot include the section being defined.

All direct entry values defined in a given section supersede any values which may have been included from another section. In the following example, `CS_OPT_ANSINULL` will be set to false in the `APP.PAYROLL` application. Note that the position of the include statement does not affect this rule.

```
[GENERIC]
  CS_OPT_ANSINULL=CS_TRUE
[APP_PAYROLL]
  CS_OPT_ANSINULL=CS_FALSE
  include=GENERIC
```

Sample programs

Consider the following scenario: An Embedded SQL program defines a cursor to retrieve rows from the titles table in the pubs2 database. The WHERE clause uses non-ANSI standard NULL checking. To clarify, IS NULL and IS NOT NULL are ANSI standards which is the default used by Embedded SQL programs, whereas an Embedded SQL program wishing to use = NULL or != NULL will need to turn OFF ANSINULL behavior and use Transact-SQL syntax instead.

In the following example, no change is made to the Embedded SQL code, but the desired behavior is attained by setting appropriate properties in the Open Client and Open Server configuration file.

There are two versions of the same program listed below. One is to be used with the `-e` option and the other with the `-x` option.

Embedded SQL/C sample makefile on Windows

The `libsybcobct.lib` and `mfrts32.lib` libraries do not need to be included in the Embedded SQL/C sample makefile.

You must change the `CC_INCLUDE` variable in the makefile to:

```
CC_INCLUDES= -I$(SYBASE)\include
```

Note On Microsoft Windows, the command to compile all the sample programs is `nmake`, not `make`.

Embedded SQL/C sample programs

Before you build Embedded SQL/C sample programs on UNIX platforms, you must:

- Set execute permission on the *sybopts.sh* file for the file's owner:

```
chmod u+x sybopts.sh
```
- If you have not already done so, include the current directory in the search path:

```
setenv PATH .:$PATH
```

Embedded SQL program version for use with the -x option

```
/* Program name: ocs_test.cp
**
** Description : This program declares a cursor which retrieves rows
** from the 'titles' table based on condition checking for NULLS
** in the NON-ANSI style.
** The program will be compiled using the -x option which will
** use an external configuration file (ocs.cfg) based on the
** name of the application. The name of the application is
** defined at the time of INITIALIZING the application. Note that
** this is a new 11.x feature too.
*/

#include <stdio.h>

/* Declare the SQLCA */
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
    /* storage for login name and password */
    CS_CHARusername[30], password[30];
    CS_CHARtitle_id[7], price[30];
EXEC SQL END DECLARE SECTION;

/*
** Forward declarations of the error and message handlers and
** other subroutines called from main().
*/
void error_handler();
void warning_handler();
```

```
int main()
{
    int i=0 ;

    EXEC SQL WHENEVER SQLERROR CALL error_handler();
    EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
    EXEC SQL WHENEVER NOT FOUND CONTINUE ;

    /*
    ** Copy the user name and password defined in sybsqlcx.h to
    ** the variables declared for them in the declare section.
    */

    strcpy(username, "sa");
    strcpy(password, "");

    EXEC SQL INITIALIZE_APPLICATION APPLICATION_NAME = "TEST1";

    EXEC SQL CONNECT :username IDENTIFIED BY :password ;
    EXEC SQL USE pubs2 ;

    EXEC SQL DECLARE title_list CURSOR FOR
    SELECT title_id, price FROM titles
           WHERE price != NULL;

    EXEC SQL OPEN title_list ;
    for ( ;; )
    {
        EXEC SQL FETCH title_list INTO
                :title_id, :price;
        if ( sqlca.sqlcode == 100 )
        {
            printf("End of fetch! \n");
            break;
        }
        printf("Title ID : %s\n", title_id );
        printf("Price      : %s\n", price) ;
        printf("Please press RETURN to continue .. ");
        getchar();
        printf("\n\n");
    }
    EXEC SQL CLOSE title_list;
    exit(0);
}
```

```
void error_handler()
{
    . . .}

void warning_handler()
{
    . . .}
```

Note Precompiler option to set in the makefile: `cpre -x`.

The following is a sample configuration file for the preceding program:

```
[DEFAULT]
;

[TEST1]
;This is name of the application set by INITIALIZE_APPLICATION. ;Therefore this
is the section that will be referred to a runtime.

CS_OPT_ANSINULL=CS_FALSE

;The above option will enable comparisons of nulls in the NON-ANSI
;style.
```

Same Embedded SQL program with the `-e` option

```
/* Program name: ocs_test.cp
**
** Description : This program declares a cursor which retrieves rows
** from the 'titles' table based on condition checking for NULLS
** in the NON-ANSI style.
** The program will be compiled using the -e option which will
** use the server name that the application connects to, as the
** corresponding section to look up in the configuration file.
*/

#include <stdio.h>

/* Declare the SQLCA */
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
    /* storage for login name and password */
```



```
        CS_CHARusername[30], password[30];
        CS_CHARtitle_id[7], price[30];
EXEC SQL END DECLARE SECTION;

/*
** Forward declarations of the error and message handlers and
** other subroutines called from main().
*/
void    error_handler();
void    warning_handler();

int main()
{
    int i=0 ;

    EXEC SQL WHENEVER SQLERROR CALL error_handler();
    EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
    EXEC SQL WHENEVER NOT FOUND CONTINUE ;

    /*
    ** Copy the user name and password defined in sybsqlcx.h to
    ** the variables declared for them in the declare section.
    */

    strcpy(username, "sa");
    strcpy(password, "");

    EXEC SQL CONNECT :username IDENTIFIED BY :password ;
    EXEC SQL USE pubs2 ;

    EXEC SQL DECLARE title_list CURSOR FOR
        SELECT title_id, price FROM titles
           WHERE price != NULL;

    EXEC SQL OPEN title_list ;
    for ( ;; )
    {
        EXEC SQL FETCH title_list INTO
            :title_id, :price;
        if ( sqlca.sqlcode == 100 )
        {
            printf("End of fetch! \n");
            break;
        }
        printf("Title ID : %s\n", title_id );
    }
}
```

```
        printf("Price      : %s\n", price) ;
        printf("Please press RETURN to continue .. ");
        getchar();
        printf("\n\n");
    }
    EXEC SQL CLOSE title_list;
    exit(0);
}

void error_handler()
{
    . . .}
}
```

Note Precompiler option to set in the makefile: `cpre -e`.

The following is a sample configuration file for the preceding program:

```
[DEFAULT]
;

[SYBASE]
;This is name of the server that the application connect to. Therefore
;this is the section that will be referred to a runtime.
;
CS_OPT_ANSINULL=CS_FALSE
;The above option will enable comparisons of nulls in the NON-ANSI
;style.
```

The above configuration files have been vastly simplified. A typical Open Client and Open Server configuration file would be in the following format:

```
[DEFAULT]
;
[ANSI_ESQL]
CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS
CS_EXTRA_INF=CS_TRUE
CS_ANSI_BINDS=CS_TRUE
CS_OPT_ANSINULL=CS_TRUE
CS_OPT_ANSIPERM=CS_TRUE
CS_OPT_STR_RTRUNC=CS_TRUE
CS_OPT_ARITHABORT=CS_FALSE
CS_OPT_TRUNCIGNORE=CS_TRUE
CS_OPT_ISOLATION=CS_OPT_LEVEL3
CS_OPT_CHAINXACTS=CS_TRUE
CS_OPT_CURCLOSEONXACT=CS_TRUE
```

```
CS_OPT_QUOTED_IDENT=CS_TRUE
;
;The following is a sample section showing how to alter standard
;configuration:
;
[RELEVANT_SECTION_NAME]
;
;Use most of the ANSI properties defined above,
;
include=ANSI_ESQL

;but override some default properties

CS_OPT_ANSINULL=CS_TRUE      ; enable non-ansi style null comparisons
CS_OPT_CHAINXACTS=CS_FALSE  ; run in autocommit mode
```

Summary

The Open Client and Open Server configuration file serves as a single location where environment settings can be managed for multiple Embedded SQL applications. The default name of this file is *ocs.cfg*, and is located in the *\$\$SYBASE/\$SYBASE_OCS/config* directory. The use of the configuration file is regulated by the use of the *-x* and *-e* precompiler options. The syntax used for modifying the Open Client and Open Server configuration file matches the existing syntax for Sybase localization and configuration files supported by CS-Library with minor variations.

Precompiler Warning and Error Messages

The Embedded SQL precompiler generates the informational, warning, and error messages in different tables.

Each table contains four fields.

- “Message_ID” lists the identification code of the message you may receive.
- “Message Text” lists the online text associated with the message you may receive.
- “Severity” lists the seriousness of the message you may receive. A message can be:
 - Information – no error or warning was detected, and the precompiler succeeded. The message is purely informational.
 - A warning – a noncritical error was detected, but the program precompiled.
 - Severe – an error occurred, and no code was generated. The precompilation failed.
 - Fatal – a severe error occurred from which the precompiler cannot recover. No further attempt will be made to process your files. Precompiler exits.
- The fourth field, “Fix,” suggests a means of correcting the situation that caused the error or warning.

Table A-1: Command line option messages

Message ID	Message text	Severity	Fix
M_COMPAT_INFO	Compatibility mode specified.	Information	No fix required.
M_DUPOPT	Duplicate command line option specified.	Severe	Do not duplicate the options specified on the command line. Remove the offending duplicate option.

Message ID	Message text	Severity	Fix
M_EXCFG_OVERRIDE	The switch value will have no effect because the external switch value has been specified.	Warning	When you use an external configuration file, you may override configuration options set on the command line. Choose one means of setting options.
M_INVALID_COMPAT	Unrecognized compatibility mode specified.	Information	No fix required.
M_INVALID_FILE_FMT	Invalid character in file value at line value.	Severe	Check that characters in the input file are valid. Also, check that you have correctly set the character set you want to use.
M_INVALID_FIPLEVEL	Invalid FIPS level specified.	Severe	Legal values are SQL92E and SQL89.
M_INVALID_SYNLEVEL	Invalid syntax checking level specified.	Severe	Legal values are NONE, SYNTAX, SEMANTIC.
M_INVLD_HLANG	Host Language specified is invalid.	Severe	Valid options are ANSI_C, KR_C.
M_INVLD_OCLIB_VER	The Open Client Client-Library version is invalid.	Severe	The correct version string is "CS_VERSION_xxx," where xxx is your current version.
M_INVOPT	Option is invalid.	Severe	Invalid option specified. Substitute the correct value.
M_LABEL_SYNTAX	Security label is improperly specified; the proper format is 'labelname=labelvalue'.	Severe	Use the allowed syntax.
M_MSGINIT_FAIL	Error initializing localized error messages.	Warning	Verify that the Sybase installation is complete and that there is a valid entry for the LANG variable in the <i>locales.dat</i> file.
M_MULTI_IN_USE_DEF_OUT	When precompiling multiple input files, you cannot specify output (Listing, SQL, or Language) file names.	Severe	Remove all -G, -L, and -O flags from the command line or precompile the files one at a time.

Message ID	Message text	Severity	Fix
M_NO_INPUT_FILE	Error: No input file is specified to be precompiled.	Severe	Specify an input file for precompilation. Note This error may occur if you precede the input file name with a flag (such as -G, for generate stored procedures) which takes an optional argument. To fix, put another flag in front of the input file name. For example, replace <code>cpre -G file.pc</code> with <code>cpre -G -Ccompilename</code> .
M_OPEN_INCLUDE	Unable to open the specified include file <i>file</i> .	Severe	The specified file is either not in the path or is missing the required read permission. Specify the path with the -I flag, and verify the read permission.
M_OPEN_INPUT	Unable to open the specified input file <i>file</i> .	Severe	Check the validity of the path and file name specified. If the file name extension is not provided, the precompiler searches for the default extension.
M_OPEN_ISQL	Unable to open the specified ISQL file <i>file</i> .	Severe	Check the validity of the isql file name (the file in which the stored procedures are written). Verify that you have write permission in the directory where the file is being created.
M_OPEN_LIST	Unable to open the specified listing file <i>file</i> .	Severe	Check the validity of the listing file name. Verify that you have write permission in the directory where the file is being created.
M_OPEN_TARGET	Unable to open the specified target file <i>file</i> .	Severe	Check the validity of the output file name. Verify that you have write permission in the directory where the file is being created.
M_OPT_MUST_BE_PROVIDED	Option <i>value</i> must be provided.	Severe	Provide a value for option.
M_OPT_REINIT	Warning: <i>value</i> switch initialized multiple times.	Warning	The specified switch has been initialized multiple times. The second and subsequent values are ignored.

Message ID	Message text	Severity	Fix
M_PATH_OFI	Error: Max allowed paths for "INCLUDE" files is 64 (OVERFLOWED).	Severe	The maximum allowed paths on the command line have been exceeded. Reduce the number of directories from which the "INCLUDE" files are fetched.
M_STATIC_HV_CNAME	Static cursor names cannot be host-variables: <i>line</i> .	Severe	Replace the host variable with a SQL identifier.
M_UNBALANCED_DQ	Unbalanced quotes in delimited identifier.	Severe	Balance the quote.

Table A-2: First pass parser messages

Message ID	Message text	Severity	Fix
M_64BIT_INT	Warning: 64 bit integer host variables are not supported. Line <i>value</i> .	Warning	Use some other host variable type (float, numeric, or 32-bit integer) and, if necessary, copy the value between the host variable and the 64-bit program variable.
M_BLOCK_ERROR	Non-matching block terminator in <i>value</i> at line: <i>value</i> .	Severe	Correct your program syntax.
M_CONST_FETCH	Error: Attempted fetch into CONST storage class variable <i>value</i> .	Severe	You cannot fetch into a constant type. To fetch the value, remove the constant qualifier in its declaration.
M_DUP_HV	Duplicate host variable in <i>file</i> at line <i>line</i> .	Severe	Another host variable with the same name is already declared in the same block. Verify that each variable within a given block has a unique name.
M_DUP_STRUNION	Duplicate structure/union in <i>file</i> at line <i>line</i> .	Severe	Another structure with the same name is already being declared in the same block. Verify that each variable within a given block has a unique name.
M_IDENT_OR_STRINGVAR	Error: item must be a SQL-identifier or a string-type variable.	Severe	Verify that the connection, cursor, or statement name is of type string or SQL identifier.

Message ID	Message text	Severity	Fix
M_IDENT_TOO_LONG	Error: Identifier <i>value</i> is too long (<i>value</i> bytes). Maximum size allowed is <i>value</i> bytes.	Severe	Ensure that the identifier length is within the allowed limit. For SDK 15.0 and later, the maximum length of an identifier is 255 characters. For earlier versions, the maximum length is 132 characters.
M_ILL_LITERAL_USAGE	Error: Use of literal parameters to an RPC with an OUTPUT qualifier is not legal.	Severe	Do not use a literal as an OUTPUT parameter to a stored procedure.
M_ILL_PARAM_MODE	Error: Mixing calling modes in an rpc call in <i>file</i> at line <i>line</i> .	Severe	Call the stored procedure with arguments passed by name or by position. Mixing these modes in the same call is illegal.
M_INDICVAR	Error: item must be an indicator-type variable.	Severe	Use a short integer.
M_INTVAR	Error: item must be an integer-type variable.	Severe	Use an integer.
M_MISMATCHED_QUOTES	Error: mismatched quotes on hex literal <i>value</i> .	Severe	Make quotes match.
M_MULTIDIM_ARRAY	Error: at line <i>line</i> . Multiple-dimensioned array variables are not supported.	Severe	Multiple-dimensioned arrays are not supported. Break up a $m \times n$ array into m arrays of n elements each.
M_MULTI_RESULTS	Error: Embedded Query at line <i>line</i> returns multiple result sets.	Severe	Break the query into multiple queries, each returning one result set. Alternatively, rewrite the queries to fill a temporary table with all the values, then select from the temporary table, thus giving a single result set.
M_NODCL_NONANSI	Warning: Neither SQLCODE nor SQLCA declared in non-ANSI mode.	Warning	In non-ANSI mode, declare either SQLCA, SQLCODE, or both. Verify that the scope is applicable for all Embedded SQL statements within the program.
M_NOLITERAL	Error: item may not be an unquoted name.	Severe	Use a quoted name or host variable.
M_NOSQUOTE	Error: item may not be a single quoted string. Use double quotes.	Severe	Use double quotes.

Message ID	Message text	Severity	Fix
M_NOT_AT_ABLE	An “at” clause is used with a statement type which does not allow it. This occurred at line <i>value</i> .	Severe	Remove the at clause from the specified statement.
M_NUMBER_OR_INDICVAR	Error: item must be an integer or an indicator-type variable.	Severe	Use a literal integer or a short integer or CS_SMALLINT.
M_NUMBER_OR_INTVAR	Error: item must be an integer constant or an integer type variable.	Severe	Unused. May be used to raise an error if some field in the dynamic SQL statements (such as, MAX, Value <i>n</i> ,) are not an integer type or an integer constant.
M_PARAM_RESULTS	Error: Embedded Query at line <i>line</i> returns unexpected parameter result sets.	Severe	Arises only during optional server syntax checking. Determine why the query is returning parameters and rewrite it.
M_PASS1_ERR	File <i>file</i> : Syntax errors in Pass 1: Pass 2 not done.	Information	Errors in Pass 1 resulted in an aborted precompilation. Correct Pass 1 errors, then proceed.
M_PTR_IN_DEC_SEC	Warning: Pointers are not yet supported in Declare section.	Warning	
M_QSTRING_OR_STRINGVAR	Error: item must be a quoted string or a type string variable.	Severe	Verify that server name, user name, and password are either double-quoted strings or of type string.
M_SCALAR_CHAR	Error: non-array character variable <i>value</i> is being used illegally as a host variable at line <i>line</i> .	Severe	Use a character array.
M_SQLCA_IGNR	Warning: Both SQLCODE and SQLCA declared: SQLCA ignored.	Warning	Remove one of the two declarations.
M_SQLCA_WARN	Warning: An INCLUDE SQLCA seen while in ANSI mode: SQLCA ignored.	Warning	
M_SQLCODE_UNDCL	Warning: SQLCODE not declared while in ANSI mode.	Warning	Declare SQLCODE.
M_STATE_CODE	Warning: both SQLSTATE and SQLCODE declared: SQLCODE ignored.	Warning	Remove one of the two declarations.

Message ID	Message text	Severity	Fix
M_STATE_SQLCA	Warning: both SQLSTATE and SQLCA declared: SQLCA ignored.	Warning	Remove one of the two declarations.
M_STATUS_RESULTS	Error: Embedded Query at line <i>line</i> returns unexpected status result sets.	Severe	Arises only during optional server syntax checking. Determine why the query is returning status results and rewrite it.
M_STICKY_AUTOVAR	Warning: Automatic variable <i>value</i> used with sticky binds at line <i>line</i> . This may cause incorrect results or errors at runtime.	Warning	Be certain that your program logic will not allow errors in this case. Alternatively, use a static or global variable.
M_STICKY_REGVAR	Error: Register variable <i>value</i> cannot be used with sticky binds at line <i>line</i> .	Severe	Remove the register qualifier.
M_STRUCT_NOTFOUND	Structure/union definition not found in scope in <i>file</i> at <i>line</i> .	Severe	Verify that the definition of the structure or union is within the scope of the specified line.
M_SYNTAX_PARSE	Syntax error in <i>file</i> at <i>line</i> .	Severe	Check the indicated line number for a syntax error in the Embedded SQL grammar.
M_UNBALANCED_DQ	Unbalanced quotes in delimited identifier.	Severe	Balance the quotes.
M_UNDEF_ELM	Error <i>value</i> : Illegal structure/union element.	Severe	The specified element of the structure is not included in the structure definition. Correct the definition.
M_UNDEF_HV	Host variable <i>value</i> undefined.	Severe	Define the host variable in the proper place.
M_UNDEF_IV	Indicator variable <i>value</i> undefined.	Severe	Define the indicator variable in the proper place.
M_UNDEF_STR	Error structure <i>value</i> undefined.	Severe	Undefined structure on the specified line. Define the structure in the proper scope.
M_UNSUP	The <i>value</i> feature is not supported in this version.	Fatal	This feature is not supported.

Table A-3: Second pass parser messages

Message ID	Message text	Severity	Fix
M_CURSOR_RD	The cursor <i>value</i> is redefined at line <i>line</i> in <i>file</i> .	Warning	A cursor with same name has already been declared. Use a different name.

Message ID	Message text	Severity	Fix
M_HOSTVAR_MULTIBIND	Warning: Host variable was used as a bind variable <i>value</i> more than once per statement.	Warning	Do not use a host variable multiple times in a single fetch statement. You cannot fetch multiple results into one location. Client-Library causes the last value fetched to be put in the variable.
M_INVTYPE_IV	Indicator variable is an incorrect type.	Severe	The indicator variable should be of type CS_SMALLINT or of type INDICATOR.
M_INVTYPE_V	Incorrect type of indicator variable found in structure <i>value</i> .	Fatal	All indicator variables in a structure must be of type CS_SMALLINT or INDICATOR.
M_INVTYPE_VI	Mismatch between number of structure elements in the indicator structure <i>value</i> and hostvar structure <i>value</i> .	Fatal	The number of elements in an indicator structure must be the same as the number of elements in the hostvar structure.
M_INVTYPE_VII	Mismatch between number of elements in the indicator array <i>value</i> and hostvar structure <i>value</i> .	Fatal	The number of elements in an indicator array must be the same as the number of elements in the hostvar structure.
M_PARSE_INTERNAL	Internal parser error at line <i>line</i> . Please contact a Sybase representative.	Fatal	Immediately report this internal consistency parser error to Sybase Technical Support.
M_SQLCANF	'INCLUDE SQLCA' statement not found.	Warning	Add statement.
M_WHEN_ERROR	Unable to find the SQL statement 'WHENEVER SQLERROR'.	Warning	Add WHENEVER SQLERROR statement or use command line option to suppress warning and 'INTO' messages (see the <i>Open Client and Open Server Programmers Supplement</i>).
M_WHEN_NF	Unable to find the SQL statement 'WHENEVER NOT FOUND'.	Warning	Enter a WHENEVER NOT FOUND statement or use command line option to suppress warning and 'INTO' messages (see the <i>Open Client and Open Server Programmers Supplement</i>).

Message ID	Message text	Severity	Fix
M_WHEN_WARN	Unable to find the SQL statement 'WHENEVER WARNING'.	Warning	Enter a WHENEVER WARNING statement or use command line option to suppress warning and 'INTO' messages (see the <i>Open Client and Open Server Programmers Supplement</i>).

Table A-4: Code generation messages

Message ID	Message text	Severity	Fix
M_INCLUDE_PATHLEN	An included or copied file path was too long. Leaving the path off the generated file name: <i>value</i> .	Warning	Use links or move the file to a shorter path.
M_WRITE_ISQL	Unable to write to the isql file. Return code: <i>value</i> .	Fatal	Verify your permission to create and write to the isql file and in the directory. Also, verify that the file system is not full.
M_WRITE_TARGET	Unable to write to the target file. Return code: <i>value</i> .	Fatal	Verify your permission to create and write to a file in the directory where the precompiler is generating the target file. Also, verify that the file system is not full.

Table A-5: FIPS flag messages

Message ID	Message text	Severity	ANSI extension
M_FIPS_ARRAY	FIPS-flagger Warning: ANSI extension ARRAY type at <i>line</i> .	Information	Arrays. As for all FIPS messages, do not use this feature if you need to be ANSI-compliant.
M_FIPS_DATAINIT	FIPS-flagger Warning: ANSI extension Data Initialization at <i>line</i> .	Information	Data initialization.
M_FIPS_LABEL	FIPS-flagger Warning: ANSI extension ':' with label in a "WHENEVER" clause.	Information	Allowing ":" with a label in a WHENEVER clause.
M_FIPS_POINTER	FIPS-flagger Warning: ANSI extension POINTER type at <i>line</i> .	Information	The type POINTER.
M_FIPS_SQLDA	FIPS-flagger Warning: ANSI extension sqlda. (line <i>line</i>).	Information	The SQLDA structure.
M_FIPS_STMT	FIPS-flagger Warning: ANSI extension statement (line <i>line</i>)	Information	The statement at this line is an extension.

Message ID	Message text	Severity	ANSI extension
M_FIPS_TYPE	FIPS-flagger Warning: ANSI extension datatype at <i>line</i> .	Information	The specified syntax is not ANSI-compliant.
M_FIPS_TYPEDEF	FIPS-flagger Warning: ANSI extension TYPEDEF <i>line</i> .	Information	TYPEDEF.
M_FIPS_VOID	FIPS-flagger Warning: ANSI extension VOID type <i>line</i> .	Information	The type VOID.

Table A-6: Internal error messages

Message ID	Message text	Severity	Fix
M_ALC_MEMORY	Unable to allocate a block of memory.	Fatal	Check system resources.
M_FILE_STACK_OVFL	File stack overflow: Max allowed nesting is <i>value</i> .	Fatal	The file stack overflowed while trying to process the nested INCLUDE statement. Do not exceed the nested depth maximum of 32.
M_INTERNAL_ERROR	Fatal Internal Error at file <i>file</i> line <i>line</i> : Argument inconsistency error. Please contact Sybase representative.	Fatal	This is an internal error. Contact your Sybase representative.

Table A-7: Sybase and Client-Library messages

Message ID	Message text	Severity	Fix
M_COLMCNT	The bind count of the <i>bind variable count</i> and the column count of result set are incompatible.	Warning	The number of returned columns is different from the number of results columns returned with the bind variable types and number.
M_COLVARLM	The host variable <i>name</i> length <i>value</i> is less than the column length of <i>value</i> .	Warning	The host variable may not be able to hold the fetched column. Check the column length and adjust the length of the host variable accordingly.
M_COLVARPS	The host variable <i>name</i> precision and scale: <i>value</i> are different from the column's precision <i>value</i> and scale: <i>value</i>	Warning	The precision and scale of the host variable is different from that of the column being fetched or inserted into. Make the scale and precision compatible.
M_COLVARTM	Open Client unable to convert type <i>value</i> to type <i>value</i> for host variable name.	Warning	Illegal type. Use <code>cs_convert</code> , as Open Client will not convert by default.
M_CTMSG	Client Library message: <i>value</i> .	Information	None. If needed, contact Sybase Technical Support for assistance.

Message ID	Message text	Severity	Fix
M_OCAPI	Error during execution of the Open Client API <i>value</i> . Error: <i>value</i> .	Warning	Depending on the context in which this warning occurs, you may be required to take corrective action before proceeding.
M_OPERSYS	Operating system error: <i>value</i> occurred during execution of the Open Client API.	Warning	An operating system error occurred. Speak with your system administrator.
M_PRECLINE	Warning(s) during check of query on line <i>value</i> .	Information	Examine the query for problems.
M_SYBSERV	Sybase server error. Server: <i>value</i> . Message: name.	Warning	Check the syntax of the statement sent to the server that caused this error. Verify that all resources are available in the Server to process the SQL statement.

Table A-8: Runtime messages

SQLCODE value, SQLSTATE code	Message text	Severity	Fix
-25001 ZZ000	Unrecoverable error occurred.	Fatal	Immediately report this error to Sybase Technical Support.
-25002 ZA000	Internal error occurred.	Fatal	Immediately report this error to Sybase Technical Support.
-25003 ZD000	Unexpected CS_COMPUTE_RESULT received.	Severe	Embedded SQL cannot retrieve compute results. Rewrite the query so it does not return them.
-25004 ZE000	Unexpected CS_CURSOR_RESULT received.	Severe	Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentations for details.
-25005 ZF000	Unexpected CS_PARAM_RESULT received.	Severe	Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.
-25006 ZG000	Unexpected CS_ROW_RESULT received.	Severe	Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.
-25007 ZB000	No message(s) returned for SQLCA, SQLCODE, or SQLSTATE.	Information	Informational message. No action is required.

SQLCODE value, SQLSTATE code	Message text	Severity	Fix
-25008 ZC000	Connection has not been defined yet.	Severe	Enter a valid connect statement.
-25009 ZH000	Unexpected CS_STATUS_RESULT received.	Severe	Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.
-25010 ZI000	Unexpected CS_DESCRIBE_RESULT received.	Severe	Verify that the value returned by the CS_LIBRARY routine is valid. Consult your CS-Library documentation for details.
-25011 22005	Data exception—error in assignment of item descriptor type.	Severe	Enter a valid descriptor type.
-25012 ZJ000	Memory allocation failure.	Severe	There is an insufficient amount of memory to allocate to this operation.
-25013 ZK000	Adaptive Server Enterprise must be version 10 or greater.	Severe	Verify that your installation has an installed, functioning copy of Adaptive Server Enterprise 10.0 or higher. If you do not have Adaptive Server Enterprise 10.0 or higher, have your installation's designated person contact Sybase Technical Support.
-25014 22024	Data exception — unterminated C string.	Severe	Be sure to null-terminate all C strings.
-25015 ZL000	Error retrieving thread identification.	Severe	An internal error probably occurred. Call Technical Support.
-25016 ZM000	Error initializing Client Library.	Severe	Check your \$SYBASE directory setup.
-25017 ZN000	Error taking a mutex.	Severe	Unused.
-25018 08002	Connection name in use.	Severe	Check your program logic: Are you re-opening an open connection? Or, use a new name for the second connection.

Note You cannot have two "DEFAULT" connections.

SQLCODE value, SQLSTATE code	Message text	Severity	Fix
-25029 ZO000	HA FAILOVER has occurred.	Information	No action required.



Sample Code for Handling Large Text and Image Data

Where to find other samples

This appendix contains sample programs for Embedded SQL that demonstrates the use of host variables in handling large text and image data. You can find additional sample programs in the Technical Documents collection of Technical Library on the Web. To access the Technical Library Web site, go to support.sybase.com, then go to the Support Services tab and select the link to "Technical Documents." Search the collection for these TechNote titles:

- Client-Library Sample Programs
- Embedded SQL/C Sample Programs
- Embedded SQL/COBOL Sample Programs

text_image.sql

Use this script to create the table called "text_tab," which you will use to run the sample program in the following section:

```
use tempdb
go

if exists (select 1 from sysobjects
where name = 'text_tab' and type = 'U' )
drop table text_tab
go

create table text_tab (
text_col text null,
image_col image null)
go
```

text_image.cp

```
/* Program name: text_image.cp
**
** Description: Inserting text and image data using host
** variables of types CS_TEXT and CS_IMAGE.

** Notes: This is a new feature in 11.x which allows you
to use
** host variables of type CS_TEXT and CS_IMAGE in insert
** or update statements to handle text or image data.
You don't
** need to use to mixed-mode client-library programming
or
** dynamic sql, which had a limit of 64 k bytes.
** The size of the text or image data that can now be
sent is
** limited only by memory or the maximum size allowed
for
** text and image data by the Adaptive Server
Enterprise. However,
** the larger the data being sent this way, the slower
the
** performance.
**
** Script file: text_image.sql
**
** Notes: Make sure you compile the program using the
'-y'
** precompiler flag.
**
*/

#include <stdio.h>
#include "sybsqllex.h"

/* Declare the SQLCA */
EXEC SQL INCLUDE sqlca;

/*
** Forward declarations of the error and message
handlers and
** other subroutines called from main().
*/
void error_handler();
void warning_handler();
```

```

int main()
{
int i=0;

EXEC SQL BEGIN DECLARE SECTION;
/* storage for login name and password */
    CS_CHAR          username[30], password[30];
    CS_TEXT          text_var[10000];
    CS_IMAGE         image_var[10000];
EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR CALL error_handler();
    EXEC SQL WHENEVER SQLWARNING CALL warning_handler();
    EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*
** Copy the user name and password defined in sybsqlx.h
to
** the variables declared for them in the declare
section.
*/
strcpy(username, USER);
strcpy(password, PASSWORD);

/* Connect to the server and specify the database to use
*/
EXEC SQL CONNECT :username IDENTIFIED BY :password;

EXEC SQL USE tempdb;

/* Put something interesting in the variables. */
for (i=0; i< 10000; i++ )
    {
    text_var[i] = 'a';
    image_var[i] = '@';
    }

EXEC SQL INSERT text_tab VALUES (:text_var, :image_var);
if ( sqlca.sqlcode == 0 )
    {
    printf("Row successfully inserted! \n");
    EXEC SQL COMMIT WORK ;
    }

EXEC SQL DISCONNECT ALL;
exit(0);

```

```
    }

    /*
    ** void error_handler()
    **
    ** Displays error codes and numbers from the SQLCA and
    exits with
    ** an ERREXIT status.
    */
    void error_handler()
    {
        fprintf(stderr, "\n** SQLCODE=(%d)", sqlca.sqlcode);

        if (sqlca.sqlerrm.sqlerrml)
        {
            fprintf(stderr, "\n** Error Message: ");
            fprintf(stderr, "\n** %s", sqlca.sqlerrm.sqlerrmc);
        }

        fprintf(stderr, "\n\n");

        exit(ERREXIT);
    }

    /*
    ** void warning_handler()
    **
    ** Displays warning messages.
    */
    void warning_handler()
    {
        if (sqlca.sqlwarn[1] == 'W')
        {
            fprintf(stderr,
                "\n** Data truncated.\n");
        }

        if (sqlca.sqlwarn[3] == 'W')
        {
            fprintf(stderr,
                "\n** Insufficient host variables to store
                results.\n");
        }
        return;
    }
}
```

Glossary

Adaptive Server Enterprise	A server in Sybase's client/server architecture. Adaptive Server Enterprise manages multiple databases and multiple users, keeps track of the actual location of data on disks, maintains mapping of logical data description to physical data storage, and maintains data and procedure caches in memory.
array	A structure composed of multiple identical variables that can be individually addressed.
array binding	The process of binding a result column to an array variable. At fetch time, multiple rows' worth of the column are copied into the variable.
batch	<p>A group of commands or statements.</p> <p>A Client-Library command batch is one or more Client-Library commands terminated by an application's call to <code>ct_send</code>. For example, an application can batch together commands to declare, set rows for, and open a cursor.</p> <p>A Transact-SQL statement batch is one or more Transact-SQL statements submitted to Adaptive Server Enterprise by means of a single Client-Library command or Embedded SQL statement.</p>
browse mode	A method that DB-Library™ and Client-Library applications can use to browse through database rows, updating their values one row at a time. Cursors provide similar functionality and are generally more portable and flexible.
bulk copy	A utility for copying data in and out of databases. Also called <code>bcp</code> .
callback event	In Open Client and Open Server, an occurrence that triggers a callback routine.
callback routine	A routine that Open Client or Open Server calls in response to a triggering event, known as a callback event.
capabilities	In terms of client/server connections, determine the types of client requests and server responses permitted for that connection.

character set	A set of specific (usually standardized) characters with an encoding scheme that uniquely defines each character. ASCII and ISO 8859-1 (Latin 1) are two common character sets.
character set conversion	Changing the encoding scheme of a set of characters on the way into or out of a server. Conversion is used when a server and a client communicating with it use different character sets. For example, if Adaptive Server Enterprise uses ISO 8859-1 and a client uses Code Page 850, character set conversion must be turned on so that both server and client interpret the data passing back and forth in the same way.
client	In client/server systems, the part of the system that sends requests to servers and processes the results of those requests.
Client-Library	Part of Open Client, a collection of routines used to write client applications. Client-Library accommodates cursors and other advanced features in the Sybase product line.
code set	See <i>character set</i> .
collating sequence	See <i>sort order</i> .
command	In Client-Library, a command is a server request initiated by an application's call to <code>ct_command</code> , <code>ct_dynamic</code> , or <code>ct_cursor</code> and terminated by the application's call to <code>ct_send</code> .
command structure	A hidden Client-Library structure (<code>CS_COMMAND</code>) that Client-Library applications use to send commands and process results.
connection structure	A hidden Client-Library structure (<code>CS_CONNECTION</code>) that defines a client/server connection within a context.
context structure	A CS-Library hidden structure (<code>CS_CONTEXT</code>) that defines an application "context," or operating environment, within a Client-Library or Open Server application. The CS-Library routines <code>cs_ctx_alloc</code> and <code>cs_ctx_drop</code> allocate and drop a context structure, respectively.
conversion	See <i>character set conversion</i> .
CS-Library	Included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications.
current row	With respect to cursors, is the row to which a cursor points. A fetch against a cursor retrieves the current row.
cursor	A symbolic name that is associated with a SQL statement.

	<p>In Embedded SQL, a cursor is a data selector that passes multiple rows of data to the host program, one row at a time.</p>
database	<p>A set of related data tables and other database objects that are organized to serve a specific purpose.</p>
datatype	<p>A defining attribute that describes the values and operations that are legal for a variable.</p>
DB-Library	<p>Part of Open Client, a collection of routines for use in writing client applications.</p>
deadlock	<p>A situation that arises when two users, each having a lock on one piece of data, attempt to acquire a lock on the other's piece of data. Adaptive Server Enterprise detects deadlocks and resolves them by killing one user's process.</p>
default	<p>Describes the value, option, or behavior that Open Client and Open Server products use when none is explicitly specified.</p>
default database	<p>The database that a user gets by default when he or she logs in to a database server.</p>
default language	<ol style="list-style-type: none">1. The language that Open Client and Open Server products use when an application does no explicit localization. The default language is determined by the "default" entry in the locales file.2. The language that Adaptive Server Enterprise uses for messages and prompts when a user has not explicitly chosen a language.
Dynamic SQL	<p>A type of SQL that allows an Embedded SQL or Client-Library application to execute SQL statements containing variables whose values are determined at runtime.</p>
error message	<p>A message that an Open Client and Open Server product issues when it detects an error condition.</p>
event	<p>An occurrence that prompts an Open Server application to take certain actions. Client commands and certain commands within Open Server application code can trigger events. When an event occurs, Open Server calls either the appropriate event-handling routine in the application code or the appropriate default event handler.</p>
event handler	<p>In Open Server, a routine that processes an event. An Open Server application can use the default handlers Open Server provides or can install custom event handlers.</p>

exposed structure	A structure whose internals are exposed to Open Client and Open Server programmers. Open Client and Open Server programmers can declare, manipulate, and de-allocate exposed structures directly. The CS_DATAFMT structure is an example of an exposed structure.
extended transaction	In Embedded SQL, a transaction composed of multiple Embedded SQL statements.
FIPS	Federal Information Processing Standards. If FIPS flagging is enabled, Adaptive Server Enterprise or the Embedded SQL precompiler issue warnings when a non-standard extension to a SQL statement is encountered.
gateway	An application that acts as an intermediary for clients and servers that cannot communicate directly. Acting as both client and server, a gateway application passes requests from a client to a server and returns results from the server to the client.
hidden structure	A structure whose internals are hidden from Open Client and Open Server programmers. Open Client and Open Server programmers must use Open Client and Open Server routines to allocate, manipulate, and deallocate hidden structures. The CS_CONTEXT structure is an example of a hidden structure.
host language	The programming language in which an application is written.
host program	In Embedded SQL, is the application program that contains the Embedded SQL code.
host variable	In Embedded SQL, a variable that enables data transfer between Adaptive Server Enterprise and the application program. See also <i>indicator variable</i> , <i>input variable</i> , <i>output variable</i> , <i>result variable</i> , and <i>status variable</i> .
indicator variable	A variable whose value indicates special conditions about another variable's value or about fetched data. When used with an Embedded SQL host variable, an indicator variable indicates when a database value is null.
input variable	A variable that is used to pass information to a routine, a stored procedure, or Adaptive Server Enterprise.
interfaces file	A file that maps server names to transport addresses. When a client application calls <code>ct_connect</code> or <code>dbopen</code> to connect to a server, Client-Library or DB-Library searches the interfaces file for the server's address. Note that not all platforms use the interfaces file. On these platforms, an alternate mechanism directs clients to server addresses.

isql script file	In Embedded SQL, one of the three files the precompiler can generate. An isql script file contains precompiler-generated stored procedures, which are written in Transact-SQL.
key	A subset of row data that uniquely identifies a row. Key data uniquely describes the <i>current row</i> in an open cursor.
keyword	A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a <i>reserved word</i> .
listing file	In Embedded SQL, one of the three files the precompiler can generate. A listing file contains the input file's source statements and informational, warning, and error messages.
locale name	A character string that represents a language/character set pair. Locale names are listed in the <i>locales file</i> . Sybase predefines some locale names, but a system administrator can define additional locale names and add them to the locales file.
locale structure	A CS-Library hidden structure (CS_LOCALE) that defines custom localization values for a Client-Library or Open Server application. An application can use a CS_LOCALE to define the language, character set, datepart ordering, and sort order it will use. The CS-Library routines <code>cs_loc_alloc</code> and <code>cs_loc_drop</code> allocate and drop a locale structure.
locales file	A file that maps locale names to language/character set pairs. Open Client and Open Server products search the locales file when loading localization information.
localization	The process of setting up an application to run in a particular national language environment. An application that is localized typically generates messages in a local language and character set and uses local datetime formats.
login name	The name a user uses to log in to a server. An Adaptive Server Enterprise login name is valid if Adaptive Server Enterprise has an entry for that user in the system table <code>syslogins</code> .
message number	A number that uniquely identifies an error message.
message queue	In Open Server, a linked list of message pointers through which threads communicate. Threads can write messages into and read messages from the queue.
multi-byte character set	A character set that includes characters encoded using more than 1 byte. EUC JIS and Shift-JIS are examples of multibyte character sets.

mutex	A mutual exclusion semaphore. This is a logical object that an Open Server application uses to ensure exclusive access to a shared object.
null	Having no explicitly assigned value. NULL is not equivalent to zero, or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.
Open Server	A Sybase product that provides tools and interfaces for creating custom servers.
Open Server application	A custom server constructed with Open Server.
output variable	In Embedded SQL, a variable that passes data from a stored procedure to an application program.
parameter	<ol style="list-style-type: none">1. A variable that is used to pass data to and retrieve data from a routine.2. An argument to a stored procedure.
passthrough mode	Tabular Data Stream™ (TDS) packets between a client and a remote data source without unpacking the packets' contents.
property	A named value stored in a structure. Context, connection, thread, and command structures have properties. A structure's properties determine how it behaves.
query	<ol style="list-style-type: none">1. A data retrieval request; usually a select statement.2. Any SQL statement that manipulates data.
registered procedure	In Open Server, a collection of C statements stored under a name. Open Server-supplied registered procedures are called <i>system registered procedures</i> .
remote procedure call (RPC)	<ol style="list-style-type: none">1. One of two ways in which a client application can execute an Adaptive Server Enterprise stored procedure. (The other is with a Transact-SQL execute statement.) A Client-Library application initiates a remote procedure call command by calling <code>ct_command</code>. A DB-Library application initiates a remote procedure call command by calling <code>dbrpcinit</code>.2. A type of request a client can make of an Open Server application. In response, Open Server either executes the corresponding registered procedure or calls the Open Server application's RPC event handler.3. A stored procedure executed on a server that is different from the server to which the user is connected.

result variable	In Embedded SQL, a variable which receives the results of a select or fetch statement.
server	In client/server systems, the server is the part of the system that processes client requests and returns results to clients.
Server-Library	A collection of routines for use in writing Open Server applications.
sort order	Used to determine the order in which character data is sorted. Also called collating sequence.
SQLCA	<ol style="list-style-type: none">1. In an Embedded SQL application, a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in SQLCA.2. In a Client-Library application, a structure that the application can use to retrieve Client-Library and server error and informational messages.
SQLCODE	<ol style="list-style-type: none">1. In an Embedded SQL application, a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in SQLCODE. A SQLCODE can exist independently or as a variable within a SQLCA structure.2. In a Client-Library application, a structure that the application can use to retrieve Client-Library and server error and informational message codes.
statement	In Transact-SQL or Embedded SQL, an instruction that begins with a keyword. The keyword names the basic operation or command to be performed.
status variable	In Embedded SQL, a variable that receives the return status value of a stored procedure, thereby indicating the procedure's success or failure.
stored procedure	In Adaptive Server Enterprise, a collection of SQL statements and optional control-of-flow statements stored under a name. Adaptive Server Enterprise-supplied stored procedures are called <i>system procedures</i> .
System Administrator	The user in charge of Adaptive Server Enterprise system administration, including creating user accounts, assigning permissions, and creating new databases. On Adaptive Server Enterprise, the System Administrator's login name is "sa."
system descriptor	In Embedded SQL, an area of memory that holds a description of variables used in Dynamic SQL statements.

system procedures	Stored procedures that Adaptive Server Enterprise supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from system tables, or as mechanisms for accomplishing database administration and other tasks that involve updating system tables.
system registered procedures	Internal registered procedures that Open Server supplies for registered procedure notification and status monitoring.
target file	In Embedded SQL, one of three files the precompiler can generate. A target file is similar to the original input file, except that all SQL statements are converted to Client-Library function calls.
TDS	(Tabular Data Stream) An application-level protocol that Sybase clients and servers use to communicate. It describes commands and results.
thread	A path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.
Transact-SQL	An enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Adaptive Server Enterprise.
transaction	One or more server commands that are treated as a single unit for the purposes of backup and recovery. Commands within a transaction are committed as a group; that is, either all of them are committed or all of them are rolled back.
transaction mode	Refers to the manner in which Adaptive Server Enterprise manages transactions. Adaptive Server Enterprise supports two transaction modes: Transact-SQL mode (also called "unchained transactions") and ANSI mode (also called "chained transactions").
user name	See <i>login name</i> .

Index

Symbols

#define 24

A

Adaptive Server
 connecting to 39
 multiple connections to 41, 42
allocate descriptor 121
allow DDL in tran 124
ANSI
 dynamic SQL 74
arrays 49
 batch 52
 double-dimensional 24, 123
 indicator 49
 multiple 33
 persistent binding 114
 select into 49
 using 33
at connect_name
 named connection 128
at connection_name 43
at connection_name clause
 in exec sql statement 150
automatic variables 116

B

-b precompiler option 105
batch arrays
 fetch into 50
batches
 get diagnostics 70
 restrictions 11
 statements 11
begin transaction 70, 72

binding 67, 74
 loops 103
 persistent 101, 117
 variables 102

C

call 97
case sensitivity
 Embedded SQL 9
character array
 declaring 24
close 124
close and cursors 62
close cursor 62
colons
 indicator variables
 30
 variables 27
command line options
 precompiler 6
command structure
 persistent 105
comments
 Embedded SQL 9
commit 45
commit transaction 72, 126
commit work 71
compatibility 6, 48
 backward 4
complex definition 24
compute clause
 disallowed 173
configuration file 185
connect 39
 multiple connections 41
connection
 naming 42

Index

- connection_name 40
- connections
 - closing 128, 145
 - default 128
 - multiple 41
 - named 128
- constructs
 - valid 11
- continue 97
- conventions
 - variable 32
- conversion
 - datatype 3, 36
- COPY files 164, 165
- ct_bind routine 101
- ct_fetch routine 101
- current row 53, 59
- cursor names
 - scoping rules 11
- cursors 53, 62, 132, 133, 135, 167, 168
 - closing 62, 124
 - declaring 55
 - deleting current row 61
 - deleting rows 139
 - dynamic 131, 170
 - example 62
 - opening 57
 - persistent binding 108
 - position 59, 60
 - retrieving data 58, 59
 - scoping 53
 - updating current row 61
 - updating rows 177

D

- Data Definition Language(DDL) 75
- Data Manipulation Language (DML) 48, 75
- databases
 - accessing 39
 - pubs2 4
 - selecting rows 172
- datatype conversions 3
 - input variables 37
 - result variables 36
- datatypes 34
 - C and SQL 34
 - converting 36
 - declaring variables 34
 - list of 34
 - list of equivalent 34
- DDL
 - Data Definition Language 75
 - in tran 124
- deallocate descriptor 130
- deallocate prepare 131
- deallocated cursors
 - persistent binding 109
- declare cursor 55, 132, 133, 135
 - dynamic 132
 - persistent binding 108
 - static 133
 - stored procedure 135
 - stored procedures 69
- declare scrollable cursor 136
- declare section 19, 20
- default server
 - connecting to 40
- default transaction mode 70
- delete 61
 - positioned cursor 138
 - searched 139
 - where current of 80
 - with cursors 61
- describe input 141
- describe output 143
- directories
 - searches 14
- disconnect 44, 145
- DML
 - Data Manipulation Language 75
- documentation
 - online 66
- double-dimensional array 24
- DSQUERY environment variable 128
- dynamic
 - parameter markers 78
- dynamic cursors
 - persistent binding 108
- dynamic parameter markers 74, 153, 168, 171
- dynamic SQL 2, 73, 131, 154, 155, 170

- method 1 76, 77
- method 2 77
- method 3 80, 83
- method 4 83, 87
- overview 73
- prepare and execute 153, 171
- prepare and fetch 171
- protocol 75
- statement 75
- stored procedures 75

E

- efficiency 101
- Embedded SQL ix, 1, 2
 - advantages 2
 - creating a program 4
 - definition 1
 - rules 9
 - sample program 8
 - syntax-checking statements 99
- environment variables 128
 - SYBASE 128
- error handler
 - writing 98
- error_hndl 98
- error-handling
 - and warning-handling routines 98
 - routines 98
- errors
 - failure to detect 99
 - precompiler-detected 99
 - SQLSTATE 18
 - testing 3
 - testing for 94
 - trapping 95, 97
- error-testing 94
- exec 147
- exec sql 150
- exec statements
 - binding 105
- execute 152
- execute immediate 76, 154
 - dynamic SQL 92
- extended transaction 72

- external 34
- external configuration file 185

F

- features and enhancements 2
 - compatibility 6
- fetch 58, 59, 155
 - within a loop 59
- fetch into 33
- fetch scrollable cursor data 158
- files
 - directory 14
 - isql 69
 - listing 96
 - multiple 6
 - precompiler-generated 6

G

- get descriptor 158
- get diagnostics 70, 161
 - batches 70
 - using 97
- go to 97

H

- handlers
 - error and warning 98
- host input variables 28
- host output variables 29
- host status variables 29
- host variables 2, 30
 - assigning data to 58
 - character string 32
 - datatypes 36
 - declaring 19
 - in fetch 58, 60
 - naming 32
 - persistent binding 102
 - scope 116
 - using 26

Index

using indicator variables 29

I

identifiers

Embedded SQL 10

implementation limit 23

include 14, 164, 165

filename 162

include file directory 14

include SQLCA 164, 165

indicator arrays 49

indicator variable

host variables example 30

indicator variables

colons

30

declaring 19

input variables 31

output and result variables 30

using 29, 33

input variables 28

converting datatypes 37

host 28

insert statements

binding 103

interactive SQL 69

interfaces file 128

invalid statements

print 48

readtext 48

writetext 48

ISO

dynamic SQL 74

isql 5

file 6, 69

K

keywords

and variable names 32

Embedded SQL 10

L

labels 181

variable 39

library

Client-Library 6

listing file 6, 96

logical names 128

M

markers

dynamic parameter 153, 168, 171

multiple arrays 33

multiple connections 41

multiple source files 6

multiple SQLCAs 14

N

named connections 128

naming conventions

variables 10

nesting

stored procedure 68

null

input value 31

null password

specifying 128

O

ocs.cfg file 185

online sample programs 66

open 57, 167

dynamic cursor 167

static cursor 168

open cursor statement

persistent binding 108

output 67

output file 69

output variables 29

P

- p precompiler option 105
- parse 5, 99
- password 40
 - null 128
- performance
 - persistent binding 101
- persistent binding 117
 - commands that cannot use 107
 - cursors 108
 - guidelines 113
 - non-cursor statements 107
 - programs that benefit 104
 - scope 105
 - subscripted arrays 114
- persistent command structure 105
- placement
 - Embedded SQL statements 9
- precompiler
 - binding options 105
 - command line options 6
 - diagnostics 99
 - dynamic SQL statements 92
 - functionality 5, 6
- precompiler options
 - binding 104
- precompiler-detected errors 98
- prepare 170
- prepare and execute 77, 78, 153
 - dynamic SQL 92
- prepare and fetch
 - dynamic SQL 92
- prepare and fetch with System Descriptor
 - dynamic SQL 92
- procedure_name 67
- program
 - creating 4
- pubs2 database 4

Q

- question mark
 - dynamic parameter marker 74
- quotation marks
 - Embedded SQL 10

R

- rebind/norebind clause 114
- reserved words
 - Embedded SQL 10
 - variable names 32
- result variables 28
 - converting datatypes 36
 - host 28
- return code 13, 16
 - SQLCODE 17
 - testing 3
- rollback
 - Adaptive Server triggers 72
 - in a trigger 70
 - transaction 172
 - work 71
- routines
 - error- and warning-handling 98
- rows
 - current 59
 - deleting 138
- rules
 - Embedded SQL 9

S

- sample programs
 - online 66
- scope
 - host variables 116
 - p and -b precompiler options 106
- scoping 11, 14
 - cursor, rules 53
 - cursors 53
 - rules 11, 33
 - SQLCA, SQLCODE, and SQLSTATE 13
- scrollable cursors
 - declaring 56
 - retrieving data 59
- select 11, 172
 - returning multiple rows 52, 59
 - returning single rows 49
 - syntax 49
 - with cursors 132, 133, 135, 155
- select clause 69

Index

- select statements
 - binding 105
 - server 40
 - connecting to 39
 - set connection 41, 174
 - set descriptor 175
 - source files
 - multiple 6
 - SQL descriptors
 - persistent binding 107
 - SQL2 standard
 - dynamic SQL 74
 - SQLCA 16
 - accessing variables 15
 - Adaptive Server-related variables 15
 - declaring 14
 - list of variables 15
 - multiple 14
 - setting variables 13
 - table 15
 - variables 14, 15
 - SQLCODE
 - fetch 157
 - multiple row selects 48
 - setting variables 13
 - stand-alone 16
 - table values 17
 - values 17
 - within SQLCA 16
 - sqlcode 94, 95
 - in error-testing 94
 - return values 94
 - SQLDAs
 - persistent binding 107
 - SQLSTATE
 - codes and error messages 18
 - setting variables 13
 - using 17
 - sqlwarn 94
 - flags 94
 - stack variables 116
 - statement batches 11
 - statement labels
 - with whenever 181
 - statements
 - dynamic SQL 85, 92
 - Embedded SQL 9
 - static cursors
 - persistent binding 108
 - status variables 28, 29
 - host 28
 - status_variable 67
 - stop 97
 - stored procedures 2, 6, 47, 66
 - declare cursor 69
 - definition 47
 - dynamic SQL 75
 - executing 66
 - parameters 66
 - return status variables 66
 - types of 66
 - subscripted arrays
 - persistent binding 114
 - SYBASE
 - environment variable 128
 - syntax checking
 - of Embedded SQL statements 99
 - system variables 15, 16, 19
- ## T
- tables
 - deleting rows 138
 - target file 6
 - testing conditions
 - whenever 96
 - thread exit 177
 - transaction
 - extended 72
 - transaction mode
 - ANSI 71
 - default 70
 - Transact-SQL 70
 - transactions 70, 126
 - ANSI 70
 - ISO 70
 - restricted statements 72
 - rolling back 172
 - Transact-SQL
 - invalid keywords in Embedded SQL 3, 48
 - keywords in Embedded SQL 10

- using Embedded SQL 47
- Transact-SQL statements 138, 147, 172, 177
- triggers 70
- typedefs 22

U

- update 61, 177
 - protocol 61
 - with cursors 61
- user 39

V

- value
 - stored procedures 67
- variables 19
 - assigning data to 59
 - datatypes 34, 37
 - declare section example 20
 - declaring 19, 20, 34
 - host 3, 30
 - host input 28
 - host result 28
 - host status 29
 - indicator 19
 - input 19, 27, 28
 - naming conventions 10, 32
 - precompiler 11
 - status 29
 - system 15, 19
 - using 27

- canceling 181
- scope of 181
- scoping rules 11
- testing conditions 95
- whenever...continue 96
- where current of 139, 157

W

- warning- and error-handling routines 93, 98
- warning handler
 - writing 98
- warning_hndl 98
- warning-handling routines 98
- warnings
 - testing for 94, 95
- whenever 94, 95, 96, 179

