# **Programming Embedded SQL for C**

Microsoft Embedded SQL for C (ESQL/C) offers programmers an alternative to writing Microsoft® SQL Server<sup>™</sup> 2000 client applications with the DB-Library for C or Open Database Connectivity (ODBC) application programming interfaces (APIs). ESQL/C enables you to incorporate Transact-SQL statements into your C-language programs.

ESQL/C is mainly used for porting your existing applications from other databases to SQL Server.

## **Getting Started with Embedded SQL for C**

In this topic, you will find Embedded SQL for C (ESQL/C) system requirements, syntax conventions, and installation information.

### **Embedded SQL for C Syntax Conventions**

ESQL/C syntax combines SQL-92 standard Embedded SQL syntax and most of the Transact-SQL syntax. ESQL/C statements work somewhat differently from, or are in addition to, standard Transact-SQL statements.

#### **Naming Conventions**

ESQL/C keywords and statements in your programs are not case-sensitive. For example, the following ESQL/C statement fragments are equivalent:

EXEC SQL CONNECT TO exec sql connect to

However, ESQL/C cursors, prepared statements, and connection names are casesensitive. The same case must be used to declare and use these names. For example, the following fragments declare two different cursors:

DECLARE CUR\_NAME CURSOR DECLARE cur\_name CURSOR

The sort order of an instance of Microsoft® SQL Server<sup>™</sup> 2000 to which you are connecting determines whether other words are case-sensitive. Note that the hyphen (-) is not permitted in Transact-SQL identifiers, such as table and column names.

**Note** The Transact-SQL keyword *null* should not be uppercase in ESQL/C programs to avoid conflict with the C keyword NULL. Also, the ESQL/C keyword *delete* and the Transact-SQL keyword *in* should not be uppercase in applications for 32-bit Microsoft Windows® to avoid conflict with 32-bit Windows-defined constants in Windows.h.

## System Requirements for Embedded SQL for C

Using ESQL/C, you can compile and run applications on various operating systems.

#### Windows NT

In Microsoft® SQL Server<sup>™</sup> 2000, ESQL/C is supported on the Intel® platform under Microsoft Windows NT® 4.0. The specific system requirements are:

• Microsoft Windows NT 4.0 Workstation version 3.51 or later

Or

Microsoft Windows NT 4.0 Server version 3.51 or later

• Microsoft Visual C++® development system (32-bit), version 2.0 or later (version 5.0 is recommended)

Or

A 100-percent compatible compiler and linker

• Microsoft SQL Server 6.0, or later. The Ntwdblib.dll file must be version 6.0, or later. This library is installed on your system under \Devtools\lib.

#### Windows 95 or Windows 98

- Microsoft Windows® 95 or Microsoft Windows 98
- Microsoft Visual C++ development system (32-bit), version 2.0 or later (version 5.0 is recommended).

Or

A 100-percent compatible compiler and linker

• Microsoft SQL Server 6.0 or later; the Ntwdblib.dll file must be version 6.0 or later

#### Windows

- The 16-bit ESQL/C compiler and libraries are available for use with SQL Server 7.0 but are not supported. The compiler and libraries can be copied from the SQL Server compact disc. The system requirements for running ESQL/C under 16-bit Windows are:
- Microsoft Windows version 3.1 or later

Or

Microsoft Windows for Workgroups version 3.11 or later

• Microsoft Visual C++ development system (32-bit), version 5.0 (minimum version is 2.0)

Or

A 100-percent compatible compiler and linker

• Microsoft SQL Server 6.0, or later. The Msdblib3.dll file must be version 6.0 or later

#### **MS-DOS**

The 16-bit ESQL/C compiler and libraries are available for use with SQL Server 7.0 but are not supported. The compiler and libraries can be copied from the SQL Server compact disc. The system requirements for running ESQL/C under 16-bit Microsoft MS-DOS® are:

- Microsoft MS-DOS version 6.22 or later
- Microsoft Visual C++ development system (16-bit), version 1.52 or later

Or

A 100-percent compatible compiler or linker

• Microsoft SQL Server 6.0 or later

# Installing Embedded SQL for C

SQL Server Setup installs these groups of files.

Directory	File	Description
\Mssql7\Binn	Nsqlprep.exe	32-bit precompiler for
		NT® 4.0 Microsoft
		Windows @ 95 and
		Windows 98
	Salaiw32 dll	Precompiler services for
	Sqiaiw52.uli	Windows NT 4.0
		Windows 95 and
		Windows 98
	Salakw32 dll	Run-time services for
	oquarwoz.an	Windows NT 4 0
		Windows 95, and
		Windows 98
\Mssql7\DevTools\Include	Sqlca.h	SQLCA header
	Sqlda.h	SQLDA header
\Mssql7\DevTools\Lib	Caw32.lib	SQLCA library for
		Windows NT 4.0,
		Windows 95, and
		Windows 98
	Sqlakw32.lib	Run-time services import
		library for Windows NT
		4.0, Windows 95, and
		Windows 98
\Mssql7\DevTools\Samples\Esqlc	* *	C samples
\Mssql7\DevTools\Lib	Ntwdblib.lib	DB-Library used for
		communicating with SQL
		Server. This is part of DB-
		Library development tools.

## **Call-level Method**

Microsoft® SQL Server<sup>™</sup> 2000 programs, written using the DB-Library or ODBC API methods, communicate directly with SQL Server through C function calls. DB-Library or ODBC functions pass SQL statements to SQL Server and return the results of queries. The call-level method of programming requires no precompiler; however, you cannot include Transact-SQL statements in your C programs, as you can with Embedded SQL for C (ESQL/C).

For example, to use DB-Library to connect to SQL Server and execute a simple query against the **pubs** sample database requires source code similar to the following:

```
#define DBNTWIN32
#include <sqlfront.h>
#include <sqldb.h>
main()
{
 DBPROCESS *dbproc;
 LOGINREC *login;
 RETCODE r;
 dbinit();
 login = dblogin();
 if (login == NULL)
   return (1);
 DBSETLUSER(login, "my_login");
 DBSETLPWD(login, "my_password");
 dbproc = dbopen(login, "my_server");
 dbfreelogin(login);
 if (dbproc == NULL)
   return (1);
 dbuse(dbproc, "pubs");
```

```
dbcmd(dbproc,
```

```
"select au_fname from authors where au_lname = 'White'");
 r = dbsqlexec(dbproc);
 if (r == FAIL)
   return (1);
 while (1)
 {
   r = dbresults(dbproc);
   if (r == SUCCEED)
   {
     /* Process the rows with dbnextrow() */
   }
   if ((r == FAIL) || (r == NO_MORE_RESULTS))
     break;
 }
 return (0);
}
```

The DB-Library approach, using C function calls, is more verbose and more flexible than the ESQL/C approach, and it can be loosely coupled to any database structure. Because a great deal of the program's behavior can be changed dynamically, DB-Library programs are often general-purpose applications. DB-Library is well-suited for environments where the database structure is not known in advance.

# **Embedded SQL Method**

Embedded SQL for C (ESQL/C) programs require preprocessing by a precompiler. The ESQL/C precompiler converts Embedded SQL statements in the program into function calls that can be accepted by a C compiler. The C compiler can then compile the resulting source code into an executable program.

For example, the following Embedded SQL code does the same task as the DB-Library example in Call-level Method:

```
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char first_name[50];
    char last_name[] = "White";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO my_server.pubs
    USER my_login.my_password;
    EXEC SQL SELECT au_fname INTO :first_name
      from authors where au_lname = :last_name;
    return (0);
}
```

Note that each Embedded SQL statement starts with the introductory expression EXEC SQL. This expression tells the precompiler that the code entered between EXEC SQL and the semicolon (;) contains Embedded SQL statements.

The ESQL/C approach, using programming statements similar to Transact-SQL, is more concise than the call-level method approach and is tightly coupled to the existing database structure. Because SQL statements are directly included in the C source code, ESQL/C programs are usually special-purpose applications. ESQL/C is well-suited for environments where the C programmer is also in control of the database structure. However, ESQL/C is less flexible in environments where the database structure is changing or is not predictable.

Generally, ESQL/C is used for porting your existing Embedded SQL application code to SQL Server with minimum modifications.

## **Embedded SQL Programming**

Embedded SQL for C (ESQL/C) programs incorporate Transact-SQL statements into C source code. But because you cannot directly submit ESQL/C programs to a C compiler, ESQL/C source programs must be processed by a precompiler that produces source code acceptable to a C compiler.

ESQL/C programming is a multistep development process that converts your original Embedded SQL source code into a Microsoft® SQL Server<sup>™</sup> 2000 application that is an executable file compiled for the appropriate operating system.

### **Embedded SQL Steps**

Embedded SQL for C (ESQL/C) programming operates as follows:

- 1. At a command prompt, the name of the ESQL/C source file and the appropriate build parameters are submitted to **nsqlprep**, which is the ESQL/C precompiler for Microsoft® Windows NT® 4.0, and Microsoft Windows® 95/98 and later operating systems. The precompiler parses the submitted file, finds the Embedded SQL statements included in the code, and processes the statements.
- The precompiler produces a C source code file with the Embedded SQL statements removed (commented out) and, if appropriate, a bind file. The Embedded SQL statements are replaced by calls to the runtime library (Sqlakw32.dll). The run-time library calls DB-Library (Ntwdblib.dll) to access servers running Microsoft SQL Server<sup>™</sup> 2000 across a network.

During this step, you can specify that stored procedures be created automatically and stored on a specific instance of SQL Server or saved in a bind file. Bind files are Transact-SQL scripts that have a .bnd extension.

- 3. The C source code file is compiled with a supported C compiler to produce an object code file.
- 4. The object code file and library routines are linked together with a supported linker to produce an executable file.

For more information about building an ESQL/C application, see <u>Building</u> <u>Applications</u>.

### **Including Embedded SQL Statements**

You can include Embedded SQL statements in the portions of your C code where C functions or routines can be placed. To distinguish Embedded SQL statements from C source code, each Embedded SQL statement must begin with the introductory keyword EXEC SQL and end with a semicolon (;). An Embedded SQL statement that does not end with a semicolon usually results in **nsqlprep** compiler error -19104 "Incorrect SQL statement syntax."

You can use a backslash (\) to continue Embedded SQL strings across more than one line of source code. A single quotation mark must precede the first character of the Embedded SQL string on the first line of source code, and a single quotation mark must appear after the last character of the string on the last line of source code. For example:

#### EXEC SQL INSERT INTO TEXT132 VALUES ('TEST 192 IS THE T ULE OF THE CONTINUATION OF LINES FROM ONE LINE TO T

You can also insert C language code after the EXEC SQL keyword on the same line. For example:

#### EXEC SQL COMMIT TRAN; printf("\n");

**Note** When you include C language code after an EXEC SQL keyword on the same line, you must use the /NOLINES precompiler option.

You can also include Embedded SQL variable declaration sections in C language code where it is valid to declare variables. Use the BEGIN DECLARE SECTION and END DECLARE SECTION statements.

#### **Scope and Visibility**

The scope of Embedded SQL variable names (statements, cursors, and connections) follow the rules that apply to C variables.

#### **Connecting to a Database**

Use a CONNECT TO statement in your application to specify the Microsoft® SQL Server<sup>™</sup> name, database name, login ID, and password for the connection. You can connect to servers and databases other than those used during precompiling.

The default time-out for a database connection is 10 seconds. If the specified server does not respond to the connection request, or if the network name lookup fails, the compilation or execution suspends for approximately 10 seconds. Use the SET OPTION statement to change the connection time-out.

For more information about Embedded SQL statements, see <u>Embedded SQL</u> <u>Statements</u>.

#### **Using Static and Dynamic Statements**

Embedded SQL for C (ESQL/C) supports both static and dynamic SQL statements.

A static SQL statement is a complete Transact-SQL statement that is embedded in the program source code. Static SQL statements can be placed into stored procedures and can contain host variables.

With dynamic SQL statements, knowing the complete structure of an SQL statement before building the application is not necessary. Dynamic SQL statements allow run-time input to provide information about the database objects to query.

A dynamic SQL statement is an incomplete Transact-SQL statement, some or all of which is supplied at run time.

Dynamic SQL statements created by using the PREPARE and EXECUTE statements can contain parameter markers and host variables. Parameter markers are question marks (?) that act as placeholders for information supplied at run time. Dynamic SQL statements executed by using the EXECUTE IMMEDIATE statement must conform to Transact-SQL language rules. They cannot contain parameter markers, host variables, or keywords that pertain exclusively to ESQL/C.

#### **Static SQL Statements**

An entire Transact-SQL transaction, including variable declarations, control-offlow language, and calls to stored procedures, can be coded as a single static SQL statement.

Static SQL statements can contain C-program host variables for input values and output data. Host variables are defined by the host C application and are accessible to the C and ESQL/C sections of your application. For more information about declaring and using host variables, see <u>Using Host Variables</u> and <u>Using the SQLDA Data Structure</u>.

When an SQL statement uses input host variables, the values of these variables are inserted in the statement before the statement runs. Output host variables are filled with the values returned after the statement runs.

**IMPORTANT** Because Embedded SQL does not support alternate format rows, COMPUTE and COMPUTE BY clauses are ignored. If a statement returns multiple result sets, only the first result set is recognized; subsequent result sets are discarded. Also, if a statement returns more than one row, only the first row is recognized unless a cursor is used; subsequent rows are discarded.

At compile time, static SQL statements can be compiled as stored procedures into an access plan or executed as dynamic SQL statements. For more information about access plans and their alternatives, see <u>Access Plans and Bind Files</u> and <u>Building Applications</u>. When a static SQL statement contains only a single transaction-management command, such as BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION, or SAVE TRANSACTION, the static SQL statement is not compiled into an access plan because stored procedures cannot contain unbalanced transaction-management statements. In these cases, at run time, the application issues unmatched transaction-management statements as dynamic SQL statements.

The rules for Transact-SQL stored procedures apply to static SQL statements.

#### **Dynamic SQL Statements**

Dynamic SQL statements are not completely embedded in the source code; instead, portions are stored in program variables that can be modified at run time. Dynamic SQL statements consist of character strings that can contain question marks (?) as parameter markers, which act as place holders for input data. For example:

DELETE FROM AUTHORS WHERE au\_fname = ? AND au\_lname :

Within an application, you can use the PREPARE, EXECUTE, and EXECUTE IMMEDIATE Embedded SQL statements to process a dynamic SQL statement. In general, dynamic SQL statements are prepared by using the PREPARE statement and then executed by using the EXECUTE statement. Optionally, when no parameter markers are used, you can combine the two statements by using the EXECUTE IMMEDIATE statement.

Using dynamic SQL statements, you can write an application that prompts a user or scans a file for information (such as database object names) that is unavailable at compile time. Use dynamic SQL statements when you must build an ad hoc Embedded SQL statement.

### **Using Host Variables**

You can manage input and output for Embedded SQL statements by using host variables. Host variables are standard C-program variables that are declared in an Embedded SQL declare section by using the BEGIN DECLARE SECTION and END DECLARE SECTION statements.

Use host variables when the number of items and their data types are known at compile time. You can use host variables in static SQL statements to specify input values or to receive output values. You can also use host variables together with parameter markers in dynamic SQL statements to specify input values or to receive the output of a dynamically prepared cursor.

When a host variable name is used in an Embedded SQL statement, the variable name begins with a colon (:). This colon enables the compiler to distinguish between host variables, and tables or columns that might have the same name.

The following example is of a C program that uses host variables. The program prompts the user for an author's last name and stores the entered value in the host variable *szLastName*. The program then retrieves the author's first name from the **pubs** database and stores the result in the host variable *szFirstName*.

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

```
int main ()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char szLastName[30];
    char szFirstName[30];
    EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL CONNECT TO gizmo.pubs USER sa;
```

```
printf("Type author's last name: ");
gets(szLastName);
```

#### EXEC SQL SELECT au\_fname INTO :szFirstName FROM authors WHERE au\_lname = :szLastName;

printf("Author's first name is %s.", szFirstName);
return (0);
}

### **Declaring Host Variables**

Before you use a host variable in an Embedded SQL statement, you must declare the variable by using normal C-language syntax within an Embedded SQL host declare section. Host variable declarations start with the Embedded SQL statement BEGIN DECLARE SECTION and end with the END DECLARE SECTION statement, for example:

EXEC SQL BEGIN DECLARE SECTION; int nID; unsigned short usNumber; char szName[30]; EXEC SQL END DECLARE SECTION;

**Note** Embedded SQL limits the length of host variable names to 30 characters. In general, Embedded SQL does not provide support for new Microsoft® SQL Server<sup>™</sup> features such as 128-bit Unicode character support for identifiers. Declaring host variables with names longer than 30 characters causes an error at precompile time when Embedded SQL processes the host variable.

Host variables can be declared wherever C variables can be declared. However, you can use a structure member as a host variable. You can also use a pointer to a single array element as a host variable and index that pointer as appropriate before each use.

**Note** You may have to modify the generated C source code when pointer variables are used. The ESQL/C precompiler will give you a warning message when it cannot determine the length of a pointer host variable. This is indicated by a value -1 in the fourth parameter to the sqlasetv API call.
# **Host Variables and Null Values**

Unlike SQL, the C language does not support variables with null (unknown or missing) values. Embedded SQL enables you to store and retrieve null values from a database by using host indicator variables. Together, a host variable and its companion indicator variable specify a single SQL value. Each of the variables must be preceded by a colon (:). When a host variable is NULL, its indicator variable has the value -1. When a host variable is nonNULL, the value of the indicator variable specifies the maximum length of the host variable data.

Place indicator variables immediately after the corresponding host variable specified in the Embedded SQL statement. For example, the following embedded UPDATE statement uses a *saleprice* host variable with a companion *saleprice\_null* indicator variable:

```
EXEC SQL UPDATE closeoutsale
SET temp_price = :saleprice :saleprice_null, listprice = :oldprice;
```

In the following SELECT statement, *price nullflag* is set to -1 because the price of this book is NULL:

```
EXEC SQL
SELECT price INTO :price:price nullflag
FROM titles
WHERE au_id = "mc3026"
```

Optionally, you can precede an indicator variable with the INDICATOR keyword when using a host variable and its associated indicator variable. For example, the following embedded UPDATE statement uses the INDICATOR keyword to more easily identify the indicator variable *saleprice\_null*:

EXEC SQL UPDATE closeoutsale SET temp\_price = :saleprice INDICATOR :saleprice\_null;

If *saleprice\_null* has a value of -1 when the UPDATE statement executes, Embedded SQL will change the statement to:

#### EXEC SQL UPDATE closeoutsale SET temp\_price = null, listprice = :oldprice;

You cannot use indicator variables in a search condition. For example, you cannot use the following Embedded SQL statement:

```
EXEC SQL DELETE FROM closeoutsale
WHERE temp_price = :saleprice :saleprice_null;
```

However, you can use the following technique to search for null values:

```
if (saleprice_null == -1)
{
    EXEC SQL DELETE FROM closeoutsale
    WHERE temp_price IS null;
}
else
{
    EXEC SQL DELETE FROM closeoutsale
    WHERE temp_price = :saleprice;
}
```

## **Host Variables and Data Types**

Microsoft® SQL Server<sup>™</sup> 2000 uses different data types than the C programming language. ESQL/C must map C data types to the appropriate SQL Server data types. The following Embedded SQL code fragment shows the mapping of three host variables, declared as C data types, to their corresponding SQL Server data types:

EXEC SQL BEGIN DECLARE SECTION; int hostvar1 = 39; char \*hostvar2 = "telescope"; float hostvar3 = 355.95; EXEC SQL END DECLARE SECTION;

EXEC SQL UPDATE inventory SET department = :hostvar1 WHERE part\_num = "4572-3";

EXEC SQL UPDATE inventory SET prod\_descrip = :hostvar2 WHERE part\_num = "4572-3";

```
EXEC SQL UPDATE inventory
SET price = :hostvar3
WHERE part_num = "4572-3";
```

In the first UPDATE statement, the **department** column has the SQL Server **smallint** (integer) data type because the host variable *hostvar1* is declared as a C **int** (integer) data type. Consequently, the data types from C map directly to SQL Server.

In the second UPDATE statement, the **prod\_descrip** column has the SQL Server **varchar** (character) data type. The *hostvar2* host variable is declared as an array of the C **char** (character) data type, which maps to the SQL **varchar** data type.

In the third UPDATE statement, the **price** column has previously been assigned the SQL Server **money** data type. No data type in C corresponds to the SQL Server **money** data type. Host variables to be used with SQL Server **money** data types can be declared as C floating-point or character data types. Embedded SQL converts those host variables to and from **money** values.

**Note** Output host variables of data type **char** are padded with blanks to their full declared length, which is an SQL-92 requirement.

Input host variables of type **char** used to input binary values must have an explicitly declared length. They cannot be pointer data types.

The following example is correct:

```
char vBinaryIn[100];
```

The following example is incorrect:

```
char *vBinaryIn="ff00";
```

Be sure to carefully match the data types of your host variables to their corresponding use in Embedded SQL statements. For more information about mapping data types from the C environment to the SQL Server environment, see <u>Advanced Programming</u>.

# Using the SQLDA Data Structure

When the number or data types of host variables to be passed are unknown at compile time, use dynamic SQL statements and the SQL descriptor area (SQLDA) data structure instead of static SQL statements and host variables. You can use the SQLDA data structure to define the type of data to be passed from the database to the host variable, or vice versa. The SQLDA data structure generally includes question marks (?) for parameter markers to specify input values for prepared Embedded SQL statements. You can also use the SQLDA data structure with the DESCRIBE or PREPARE INTO statements to receive data from a prepared SELECT statement. Although you cannot use the SQLDA data structure with static SQL statements, you can use it with a cursor FETCH statement, regardless of whether the cursor is dynamic or static.

**Note** For **text** and **image** data, the maximum length of the data column is the size of *sqllen*, which is 32 KB. *Sqllen* is of the data type **short**.

# **Data Input and Output Using the SQLDA Data Structure**

The SQLDA data structure contains descriptive information about each input parameter or output column. The structure contains the column name, data type, length, and pointer to the actual data buffer for each input or output variable.

For output data, you can use the DESCRIBE statement (or the PREPARE statement with the INTO option) to enter column name, data type, and other data into the appropriate fields of the SQLDA data structure.

Before using the SQLDA structure in a PREPARE INTO or DESCRIBE statement, your application must set the length of SQLDA and the maximum number of entries.

To use the SQLDA structure for input data, your application must supply the data for the fields of the entire SQLDA data structure. If the *sqltype* field has an odd code number (value), the address of the indicator variable must also be supplied. For more information about the SQLDA data structure, see <u>Advanced Programming</u>.

# **Using Cursors**

When you write code for a transaction that retrieves a single row of results, you can use a SELECT INTO statement. This is called a singleton SELECT statement.

When you write code for a transaction where the result set includes several rows of data, you must declare and use a <u>cursor</u>. A cursor is a mechanism you can use to fetch rows one at a time. For example, if you write code that includes a SELECT statement or stored procedure that returns multiple rows, you must declare a cursor and associate it with the SELECT statement. Then, by using the FETCH statement, you can retrieve one row at a time from the result set.

You can also use cursors to perform operations within a result set. These operations are known as positioned update and positioned delete. For more information, see <u>Positioned UPDATE or DELETE Statements</u>.

**IMPORTANT** The cursor options available with Embedded SQL for C (ESQL/C) are different from the cursor options available with the Transact-SQL DECLARE CURSOR option, and the two should not be intermixed. If a Transact-SQL cursor is not available as an ESQL/C cursor option, it cannot be used.

Do not use ESQL/C cursors to process Transact-SQL batches or other operations that return multiple result sets. If a statement returns multiple result sets, only the first result set is recognized and subsequent result sets are discarded. If COMPUTE rows are returned, the rows are also ignored.

ESQL/C includes standard and browse cursor types. A standard cursor is used to retrieve one row of data at a time and shares the same connection to Microsoft® SQL Server<sup>™</sup> as the main program. Standard cursors require a unique index in SQL Server version 6.0 and earlier. To set standard cursors, use the SET CURSORTYPE CUR\_STANDARD statement or the DECLARE CURSOR statement with the FOR UPDATE option. A browse cursor is used to retrieve one row of data at a time and requires a separate connection to SQL Server. To set browse cursors, use the SET CURSORTYPE CUR\_BROWSE statement.

Standard and browse cursors are declared and used (including FETCH and positioned update or delete operations) in the same way. Standard cursors are

based on the DB-Library cursor model and allow multiple cursor operations to share the same connection to SQL Server. Each browse cursor requires a separate connection. For most applications, standard cursors are recommended and are the default because a shared single connection avoids potential locking conflicts between cursors.

Standard DB-Library cursors provides detailed descriptions of standard DB-Library cursors. DB-Library cursors have several options for controlling row membership, locking, and performance characteristics. These options are available to ESQL/C programs through the SET ANSI\_DEFAULTS, SET CURSOR\_CLOSE\_ON\_COMMIT, SET SCROLLOPTION, SET CONCURRENCY, and SET FETCHBUFFER statements. A SET option remains in effect for all cursor operations within an ESQL/C program until that option is changed by another SET statement.

Browse Cursors provides details about browse cursors. If positioned update or delete statements are used on a browse cursor, the SELECT statement used in the cursor declaration must include the FOR BROWSE option. However, because each browse cursor uses a separate database connection, SQL Server treats each cursor as a separate user. This can result in locking conflicts between different cursors in the same program. For more information, see <u>Cursors and Lock</u> <u>Conflicts</u>.

#### See Also

Standard DB-Library Cursors

Browse Cursors

# **Declaring Cursors**

To define a cursor for row-at-a-time retrieval, use the DECLARE CURSOR statement. You can declare a cursor in one of two ways: as a static cursor or as a dynamic cursor.

#### **Using Static Cursors**

For a static cursor, the complete SELECT statement is contained in the DECLARE CURSOR statement. The SELECT statement can contain host variables for input parameters. When the OPEN statement is performed on a cursor, the values of the input parameters for the host variable are read into the SELECT statement. You cannot specify host variables and SQLDA data structures in the OPEN statement for a static cursor because the input host variables are already identified in the DECLARE CURSOR statement.

This is an example of a static cursor:

```
EXEC SQL BEGIN DECLARE SECTION;
char szLastName[] = "White";
char szFirstName[30];
EXEC SQL END DECLARE SECTION;
```

EXEC SQL DECLARE author\_cursor CURSOR FOR SELECT au\_fname FROM authors WHERE au\_lname = :szLastNan

EXEC SQL OPEN author\_cursor; EXEC SQL FETCH author\_cursor INTO :szFirstName;

#### **Using Dynamic Cursors**

For a dynamic cursor, the SELECT statement is not contained in the DECLARE CURSOR statement. Instead, the DECLARE CURSOR statement references the name of a prepared SELECT statement that can contain parameter markers (?) to indicate that data must be supplied when a cursor is opened. You must declare a

dynamic cursor by using the DECLARE CURSOR statement before you prepare a SELECT statement.

When a prepared SELECT statement contains parameter markers, the corresponding OPEN statement must specify the host variables or the name of the SQLDA data structure that will supply the values for the parameter markers. The data type, length, and address fields of the specified SQLDA data structure must already contain valid data.

This is an example of a dynamic cursor:

EXEC SQL BEGIN DECLARE SECTION; char szCommand[] = "SELECT au\_fname FROM authors WHERE au\_ char szLastName[] = "White"; char szFirstName[30]; EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE author\_cursor CURSOR FOR select\_statement;

EXEC SQL PREPARE select\_statement FROM :szCommand;

EXEC SQL OPEN author\_cursor USING :szLastName; EXEC SQL FETCH author\_cursor INTO :szFirstName;

## **Positioned UPDATE or DELETE Statements**

Positioned UPDATE and DELETE statements are used in conjunction with cursors and include WHERE CURRENT OF clauses instead of search condition clauses. The WHERE CURRENT OF clause specifies the location of the corresponding cursor.

You can perform a prepared positioned update or a ROLLBACK TRANSACTION operation only on dynamic cursors.

On a cursor declaration that contains a join, you can perform a positioned update or positioned delete only on the first table in the join list.

## **Managing Transactions**

With Embedded SQL for C (ESQL/C), you can use the full transaction control facilities of Microsoft® SQL Server<sup>™</sup> 2000.

The COMMIT TRANSACTION statement marks the end of a user-defined transaction initiated by a BEGIN TRANSACTION statement. The COMMIT statement makes changes to the transaction's database permanent and visible to other users. It also removes all locks from the affected data so that other users can access the data.

As with other SQL Server applications, statements not bound by BEGIN TRANSACTION and COMMIT TRANSACTION are automatically committed when the statement executes without an error.

The ROLLBACK TRANSACTION statement reverses the effects of a userspecified transaction to the beginning of the OPEN TRANSACTION or to the last save point (marked by a Transact-SQL SAVE TRANSACTION statement) inside the open transaction. After a transaction is committed, it cannot be rolled back.

Note that by default, a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement does not close cursors and applies only to the current connection if multiple connections are active. You can use the SET CURSOR\_CLOSE\_ON\_COMMIT statement to close all cursors on a connection automatically when a COMMIT TRANSACTION or a ROLLBACK TRANSACTION statement is issued.

# Using the SQLCA Data Structure

Microsoft® SQL Server<sup>™</sup> 2000 uses the SQL communications area (SQLCA) data structure to trap and report run-time errors to your Embedded SQL for C (ESQL/C) applications. Your application can check the error fields and status indicators of the SQLCA data structure to determine the success or failure of an Embedded SQL statement. The precompiler automatically includes the SQLCA data structure in ESQL/C applications.

You can include routines in your application to test the SQLCODE, SQLWARN, SQLERRM, SQLERRD, and SQLSTATE fields of the SQLCA data structure and to provide follow-up procedures according to the status returned.

- The SQLCODE field contains the negative SQL Server error code (the ESQL/C standard requires that error codes be negative).
- The SQLWARN flags are set if certain exceptions, such as data truncation, occur.
- The SQLERRM field contains the text of the error message.
- The SQLERRD1 field contains the error number.
- The SQLERRD3 array indicates the number of rows affected.
- The SQLSTATE field contains run-time errors that generate SQL-92 standard SQLSTATE codes.

Because the character fields of SQLCA (such as SQLWARN and SQLERRMC) are FAR pointers in Microsoft Windows®, you must use the **%Fs** format specifier for them when using **printf** and similar functions.

For more information about the SQLCA data structure, see <u>Advanced</u> <u>Programming</u>.

# **SQLCODE** Variable

The most important and widely used field of the SQLCA data structure is the SQLCODE variable. Each time Microsoft® SQL Server<sup>™</sup> 2000 runs an Embedded SQL statement, it sets the value of the SQLCODE variable to indicate whether the last Embedded SQL statement completed successfully. A value of 0 indicates that the last Embedded SQL statement was successful. Values other than 0 indicate warnings or errors.

To use SQLCODE in your program, you can either declare it explicitly as long SQLCODE; or leave it undeclared and have the precompiler generate a declaration. However, the precompiler only generates a declaration in a source module that contains a **main()** or a **WinMain()** function.

The preprocessor automatically inserts the following definition in all .sqc modules, so the actual definition of SQLCODE should only occur in one module.

#### extern long SQLCODE;

If your project does not include a .sqc file that includes a **main()** or **WinMain()** function, you must explicitly declare SQLCODE in one of your modules. Explicit SQLCODE declarations can occur either inside or outside of a host variable declaration block.

#### See Also

Advanced Programming

# **SQLSTATE Variable**

ESQL/C supports SQLSTATE codes, which return errors encountered at run time. SQLSTATE codes fall into two categories: those that must be generated locally by DB-Library or ESQL/C, and those that are generated by the server. SQLSTATE codes always correspond to SQLCODE values.

#### See Also

**SQLSTATE Messages** 

# **Using the WHENEVER Statement**

Writing code to check the value of the SQLCODE variable after each Embedded SQL statement becomes burdensome, especially when writing large programs. Another method for checking the status of the SQLCA data structure fields is the WHENEVER statement. The WHENEVER statement is not an executable statement. It is a directive to the ESQL/C precompiler to generate code automatically to handle errors after each executable Embedded SQL statement, and it specifies the next action to be taken. The WHENEVER statement allows one of three actions (CONTINUE, GOTO, or CALL) to be registered for each of the three possible SQLCODE conditions (SQLWARNING, SQLERROR, or NOT FOUND).

A WHENEVER statement in the program code supersedes the conditions of all earlier WHENEVER statements.

This is an example of a WHENEVER statement:

EXEC SQL WHENEVER sqlerror GOTO errormessage1;

EXEC SQL DELETE FROM homesales WHERE equity < 10000;

EXEC SQL DELETE FROM customerlist WHERE salary < 40000;

EXEC SQL WHENEVER sqlerror CONTINUE;

EXEC SQL UPDATE homesales SET equity = equity - loanvalue;

EXEC SQL WHENEVER sqlerror GOTO errormessage2;

EXEC SQL INSERT INTO homesales (seller\_name, sale\_price) real\_estate('Jane Doe', 180000.00);

```
·
.
errormessage1:
printf("SQL DELETE error: %ld\n, sqlcode);
exit();
```

•

```
errormessage2:
    printf("SQL INSERT error: %ld\n, sqlcode);
exit();
```

For more information, see WHENEVER in Embedded SQL Statements.

## **Embedded SQL for C Reference**

The maximum size of a single Embedded SQL statement is 8,191 characters for 16-bit Microsoft® Windows® 2000 and 19,999 characters for Microsoft Windows NT® 4.0.

## **Embedded SQL Statements**

For the Embedded SQL statements listed, an asterisk (\*) identifies statements that have names identical to names of Transact-SQL statements. For statements with asterisks, the syntax included augments the standard Transact-SQL syntax.

BEGIN DECLARE SECTION	PREPARE
CLOSE*	SELECT INTO*
CONNECT TO	SET ANSI_DEFAULTS
DECLARE CURSOR*	SET CONCURRENCY
DELETE (POSITIONED)*	SET CONNECTION
DELETE (SEARCHED)*	SET CURSOR_CLOSE_ON_COMMIT
DESCRIBE	SET CURSORTYPE
DISCONNECT	SET FETCHBUFFER
END DECLARE SECTION	SET OPTION
EXECUTE*	SET SCROLLOPTION
EXECUTE IMMEDIATE	UPDATE (POSITIONED)*
FETCH*	UPDATE (SEARCHED)*
GET CONNECTION	WHENEVER
OPEN*	

# **BEGIN DECLARE SECTION**

The BEGIN DECLARE SECTION statement marks the beginning of a C host-variable declaration section.

#### Syntax

**BEGIN DECLARE SECTION** 

## Remarks

The BEGIN DECLARE SECTION statement can be included anywhere C permits declaring variables, and where declared host variables follow the normal rules for scoping in C. Use END DECLARE SECTION to identify the end of a C declaration section. The embedded BEGIN DECLARE SECTION statement must follow the EXEC SQL introductory keyword.

Declare sections cannot be nested.

Use the following rules for declaring host variables:

- Host variables must be declared in C, not in Transact-SQL.
- Host variables referenced by Embedded SQL statements must be included in a declaration section that appears before the statement.

## Examples

EXEC SQL BEGIN DECLARE SECTION; int id; char name[30]; EXEC SQL END DECLARE SECTION;

See Also

END DECLARE SECTION
# CLOSE

The CLOSE statement ends row-at-a-time data retrieval initiated by the OPEN statement for a specified cursor, and closes the cursor connection.

### Syntax

CLOSE cursor\_name

## Arguments

cursor\_name

Is a previously declared and opened cursor. Cursor names can have as many as 30 characters, and can include alphanumeric characters and any symbols that are legal in file names. Hyphens (-) are not permitted. The first character of a cursor name must be a letter.

## Remarks

The CLOSE statement discards unprocessed rows and frees any locks held by the cursor. The cursor must be declared and opened before it can be closed. All open cursors are closed automatically at the end of the program.

## Examples

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT id, name, dept, job, years, salary, comm FROM staff;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
{
    /* SQLCODE will be zero if data is successfully fetched */
EXEC SQL
FETCH c1 INTO :id, :name, :dept, :job, :years, :salary, :comm;
if (SQLCODE == 0)
```

printf("%4d %12s %10d %10s %2d %8d %8d",

id, name, dept, job, years, salary, comm); } EXEC SQL CLOSE c1;

## See Also

DECLARE CURSOR

<u>OPEN</u>

<u>FETCH</u>

SET CURSOR\_CLOSE\_ON\_COMMIT

# **CONNECT TO**

The CONNECT TO statement connects to a specific database with the supplied username and password.

### Syntax

CONNECT TO {[*server\_name.*]*database\_name*} [AS *connection\_name*] USER [*login*[.*password*] | \$integrated]

### Arguments

#### server\_name

Is the server running Microsoft® SQL Server<sup>™</sup> 2000. If you omit the *server\_name*, the local server is assumed.

#### database\_name

Is the database.

#### connection\_name

Is a name for the connection. Connection names can have as many as 30 characters, and can include alphanumeric characters and any symbols that are legal in file names. Hyphens (-) are not permitted. The first character must be a letter. Do not use *current* or *all* for the connection name; they are not supported.

#### login

Is the user's login ID.

#### password

Is the user's password.

#### \$integrated

Specifies that forced integrated security is used for run-time or compile-time applications instead of the *login* and *password*.

## Remarks

The options can include character literals or host variables. If you use only one connection, you do not need to supply a name for the connection. When you use more than one connection, you must specify a name for each connection.

Connection names are global within a process. Named connections are shared by separately compiled programs linked into a single executable module. Named connections are also shared by a program and dynamic-link libraries that execute in a single process.

All database transactions after a CONNECT TO statement that do not involve a browse cursor work through the most recently declared, current connection. To use a different connection, you must use the SET CONNECTION statement.

## Examples

EXEC SQL CONNECT TO :svr USER :usr;

Or

EXEC SQL CONNECT TO "gizmo.pubs" USER "sa";

Or

EXEC SQL CONNECT TO gizmo.pubs USER sa;

### See Also

**DISCONNECT** 

# **DECLARE CURSOR**

Defines a cursor for row-at-a-time data retrieval.

### Syntax

DECLARE cursor\_name [INSENSITIVE] [SCROLL] CURSOR FOR
{select\_stmt | prepared\_stmt\_name} [FOR { READ ONLY | UPDATE [ OF
column\_list ] } ]

## Arguments

#### cursor\_name

Is the cursor name in subsequent statements. Cursor names can have as many as 30 characters, and can include alphanumeric characters and any symbols that are legal in file names. Hyphens (-) are not permitted. The first character must be a letter. Optionally, the *cursor\_name* parameter can be enclosed in quotation marks (' ').

#### **INSENSITIVE**

Specifies creating a standard, read-only cursor that is a snapshot of the cursor result set at open time. It is equivalent to the INSENSITIVE option of the Transact-SQL DECLARE CURSOR statement.

#### SCROLL

Specifies allowing first, last, and backward fetch operations. It is equivalent to issuing the SET CURSORTYPE CUR\_STANDARD and SET SCROLLOPTION KEYSET statements.

#### select\_stmt

Is any valid Transact-SQL SELECT statement. Browse cursors can also use a stored procedure that contains a SELECT statement. This SELECT statement must not contain any aggregates.

#### prepared\_stmt\_name

Is the name of a prepared SQL SELECT statement.

#### FOR READ ONLY

Specifies the use of standard DB-Library read-only cursors. This is equivalent to issuing both the SET CONCURRENCY READONLY and the SET CURSORTYPE CUR\_STANDARD statements. Using the FOR READ ONLY option overrides the SET CONCURRENCY statement.

#### FOR UPDATE

Specifies that cursors are updatable by default; therefore, the DECLARE statement does not require a FOR UPDATE option. However, if the DECLARE statement contains the FOR UPDATE option, the effect is equivalent to issuing both the SET CONCURRENCY LOCKCC and the SET CURSORTYPE CUR\_STANDARD statements. Using the FOR UPDATE option overrides the SET CONCURRENCY statement.

### Remarks

The DECLARE CURSOR statement associates the cursor name with the specified SELECT statement and enables you to retrieve rows of data by using the FETCH statement.

Cursor names are global within a program module (source code file). Cursors cannot be shared by separately compiled programs that are linked into a single executable module, or by a program and dynamic-link libraries that run in a single process.

The DECLARE CURSOR statement must appear before the first reference of the cursor. The SELECT statement runs when the cursor is opened.

The following rules apply to the SELECT statement:

- It cannot contain an INTO clause or parameter markers (?).
- It can contain input host variables that were previously identified in a host variable declaration section.
- It must include a HOLDLOCK option to enable repeatable reads. Additionally, standard cursors require that an explicit user-defined transaction is open (opened by using BEGIN TRANSACTION).

- For a standard cursor, use the SET CURSORTYPE CUR\_STANDARD if you do not use the FOR UPDATE option.
- For a browse cursor, include the FOR BROWSE option and use the SET CURSORTYPE CUR\_BROWSE statement if positioned updates or deletes will be performed on a browse cursor. If the SET CURSORTYPE statement is not used, the FOR BROWSE option makes the cursor read-only. Do not use the FOR UPDATE option.

You must declare a dynamic cursor by using the DECLARE CURSOR statement before you prepare a SELECT statement.

### Examples

EXEC SQL DECLARE c1 CURSOR FOR SELECT au\_fname, au\_lname FROM authors FOR BROWSE;

### See Also

**BEGIN DECLARE SECTION** 

PREPARE

<u>CLOSE</u>

SELECT INTO

<u>FETCH</u>

SET CURSOR\_CLOSE\_ON\_COMMIT

<u>OPEN</u>

# **DELETE (POSITIONED)**

The DELETE (POSITIONED) statement removes the row where the cursor is currently positioned.

### Syntax

DELETE [FROM] {*table\_name* | *view\_name*} WHERE CURRENT OF *cursor\_name* 

## Arguments

FROM

Is an optional keyword included for compatibility with other versions of ESQL/C.

table\_name

Is the same table used in the SELECT statement portion of the DECLARE CURSOR STATEMENT.

view\_name

Is the same view used in the SELECT statement portion of the DECLARE CURSOR statement.

cursor\_name

Is a previously declared, opened, and fetched cursor. Cursor names can have as many as 30 characters, and can include alphanumeric characters and any symbols that are legal in file names. Hyphens (-) are not permitted. The first character must be a letter.

## Remarks

In addition to having the functionality of the Transact-SQL DELETE statement, the Embedded SQL DELETE statement includes functionality known as positioned delete, which deletes the row most recently fetched by a cursor. The DELETE statement used in standard Transact-SQL statements is known as a

searched delete.

Note that a positioned delete has no search condition. The WHERE CURRENT OF option is used in place of a search condition clause. The WHERE CURRENT OF option cannot be used in a PREPARE statement.

In a positioned delete that uses a browse cursor, the SELECT statement used to open the cursor must include a FOR BROWSE clause. The base table(s) must include a **timestamp** column. If an error prevents any row found by the search condition from being deleted, no changes are made to the database.

When using a browse cursor, or a standard cursor with optimistic concurrency control (SET CONCURRENCY with the OPTCC or OPTCCVAL option), if the row has been changed after the last FETCH statement, no changes are made to the database and the value of SQLCODE is set to -532. Also, the SQLERRD3 field in the SQLCA data structure shows that no rows were processed.

### Examples

```
EXEC SQL DECLARE c1 CURSOR FOR
 SELECT au fname, au lname FROM authors FOR BROWSE;
EXEC SQL OPEN c1;
while (SQLCODE == 0)
ł
 EXEC SQL FETCH c1 INTO :fname, :lname;
 if (SQLCODE == 0)
 {
   printf("%12s %12s\n", fname, lname);
   printf("Delete? ");
   scanf("%c", &reply);
   if (reply == 'y')
   {
    EXEC SOL DELETE FROM authors WHERE CURRENT OF c
    printf("delete sqlcode= %d\n", SQLCODE(ca));
   }
 }
}
```

See Also

DECLARE CURSOR

<u>FETCH</u>

# **DELETE (SEARCHED)**

The DELETE (SEARCHED) statement removes table rows that meet the search criteria. DELETE is a standard Transact-SQL statement.

### Syntax

DELETE [FROM] {table\_name | view\_name} [WHERE search\_conditions]

## Arguments

FROM

Is an optional keyword included for compatibility with other versions of ESQL/C.

table\_name

Is the table to remove rows from.

view\_name

Is the view to remove rows from.

search\_conditions

Is any expression that can legally follow the standard Transact-SQL WHERE clause.

## Remarks

If you do not use a WHERE clause, all rows in the table specified in the DELETE statement are removed. The table, although it no longer contains data, exists until you use a DROP TABLE statement.

You cannot use DELETE on a view with a FROM clause that specifies more than one table. This would change several tables and is not supported. However, UPDATE and INSERT statements that affect only one base table of the view are supported.

# Examples

EXEC SQL DELETE FROM authors WHERE au\_lname = 'White'

# DESCRIBE

The DESCRIBE statement populates the SQLDA data structure.

### Syntax

DESCRIBE prepared\_stmt\_name INTO :sqlda\_struct

### Arguments

prepared\_stmt\_name

Is a prepared SQL statement. For more information, see <u>PREPARE</u>.

#### sqlda\_struct

Is the output SQLDA data structure to be populated.

### Remarks

The DESCRIBE statement processes prepared, dynamic SQL statements. This statement populates the specified SQLDA data structure with the data type, the length, and the column name of each column returned by the specified prepared statement. (Prepared statements are created by using the PREPARE statement. Note that the DESCRIBE statement cannot be used in the FROM clause of a PREPARE statement.) The DESCRIBE statement works like a PREPARE statement with an INTO clause.

Before using DESCRIBE, the *sqln* and *sqlabc* fields of the SQLDA data structure that are allocated by the application should be set to appropriate values. The *sqln* field must be set to the maximum number of column descriptor entries that can be held. The *sqlabc* field must be set to the length, in bytes, of the SQLDA data structure. The length is computed by using the SQLDASIZE macro as follows:

## SQLDASIZE(mysqlda->sqln)

When the DESCRIBE statement is executed, it sets the *sqld* field of the SQLDA data structure to the number of column descriptors used by the prepared

statement. If a nonSELECT statement was prepared, *sqld* is set to 0.

If the application does not know the maximum number of column descriptors required for the prepared statement, it can set *sqln* to 0 before using the DESCRIBE statement. Then DESCRIBE will set *sqld* to the maximum number of column descriptors required without actually constructing any column descriptors.

After the DESCRIBE statement has populated the SQLDA data structure, and before it has used the FETCH statement, the application must insert the address of each output variable into the *sqldata* field (part of the *sqlvar* field) of the SQLDA data structure.

## Examples

```
#define NUM_RETURN_COLS 2
struct sqlda *mysqlda;
EXEC SQL BEGIN DECLARE SECTION;
char statement[] = "SELECT au_fname, au_lname FROM authors";
EXEC SQL END DECLARE SECTION;
```

```
mysqlda = malloc(SQLDASIZE(NUM_RETURN_COLS));
if (mysqlda == NULL)
{
    return;
}
mysqlda->sqln = NUM_RETURN_COLS;
mysqlda->sqldabc = SQLDASIZE(NUM_RETURN_COLS);
```

```
EXEC SQL DECLARE c1 CURSOR FOR stmt1;
EXEC SQL PREPARE stmt1 FROM :statement;
EXEC SQL DESCRIBE stmt1 INTO :mysqlda;
// SQLDA now contains a description of the dynamic SQL statement //
EXEC SQL OPEN c1;
EXEC SQL FETCH c1 USING DESCRIPTOR :mysqlda;
```

See Also

**PREPARE** 

Advanced Programming

# DISCONNECT

The DISCONNECT statement disconnects one or all database connections.

### Syntax

DISCONNECT [connection\_name | ALL | CURRENT]

### Arguments

connection\_name

Is the connection to be disconnected.

#### ALL

Specifies disconnecting all connections. This option must be used before you can exit the program.

#### CURRENT

Specifies disconnecting the current connection. The current connection is either the most recent connection established by a CONNECT TO statement or a subsequent connection set by a SET CONNECTION statement.

### Remarks

When a connection is disconnected, all cursors opened for that connection are automatically closed.

To ensure a clean exit, an Embedded SQL program must issue a DISCONNECT ALL statement before it exits the main application.

## Examples

EXEC SQL CONNECT TO caffe.pubs AS caffe1 USER sa; EXEC SQL CONNECT TO latte.pubs AS latte1 USER sa; EXEC SQL SET CONNECTION caffe1; EXEC SQL SELECT name FROM sysobjects INTO :name; EXEC SQL SET CONNECTION latte1; EXEC SQL SELECT name FROM sysobjects INTO :name; EXEC SQL DISCONNECT caffe1; EXEC SQL DISCONNECT latte1; // The first select takes place against the pubs // // database on server "caffe." The second SELECT will // // take place against the pubs database on server "latte." // // In place of the two "disconnect" statements at the end, // // you can also write: // // EXEC SQL DISCONNECT ALL; //

### See Also

CONNECT TO

SET CONNECTION

# **END DECLARE SECTION**

The END DECLARE SECTION statement marks the end of a declaration section for host variables.

### Syntax

END DECLARE SECTION

### Remarks

The END DECLARE SECTION statement must be preceded by a BEGIN DECLARE SECTION statement.

### Examples

EXEC SQL BEGIN DECLARE SECTION; int id; char name[30]; EXEC SQL END DECLARE SECTION;

### See Also

**BEGIN DECLARE SECTION** 

# EXECUTE

The EXECUTE statement runs a prepared embedded SQL statement.

### Syntax

EXECUTE prepared\_stmt\_name [USING DESCRIPTOR :sqlda\_struct | USING :hvar [,...]]

## Arguments

prepared\_stmt\_name

Is an SQL statement that was previously prepared.

sqlda\_struct

Is an SQLDA data structure that was previously declared and that contains a description of the input values.

hvar

Is one or more input host variables.

## Remarks

The EXECUTE statement processes dynamic SQL statements. It runs the specified prepared SQL statement after it substitutes values for any parameter markers (?) present. (Prepared statements are created by using the PREPARE statement.) Only statements that do not return results are supported.

With the USING DESCRIPTOR *:sqlda\_struct* option, the values of the program variables are substituted for parameter markers in the prepared statement. The program variables are addressed by corresponding *sqldata* entries in the SQLDA data structure. (The *sqldata* field is part of the *sqlvar* field.)

If the prepared statement contains parameter markers, the EXECUTE statement must include either the USING *:hvar* option with the same number of host variables in the same order as in the prepared statement, or the USING DESCRIPTOR *:sqlda\_struct* option that identifies the SQLDA data structure

already populated by the application.

Also, the number of parameter markers in the prepared statement must match the number of *sqldata* entries (USING DESCRIPTOR *:sqlda\_struct*) or host variables (USING *:hvar*).

### Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char stmtbuf[] = "INSERT INTO publishers VALUES (?, ?, ?, ?)";
int pubid;
char pubname[30];
char city[30];
char state[3];
EXEC SQL END DECLARE SECTION;
```

```
// Prompt the user for publication data //
printf("Enter publication ID number: ");
scanf("%d", pubid);
printf("Enter publication name: ");
scanf("%s", pubname);
printf("Enter city: ");
scanf("%s", city);
printf("Enter state: ");
scanf("%s", state);
```

EXEC SQL PREPARE stmt FROM :stmtbuf;

EXEC SQL EXECUTE stmt USING :pubid, :pubname, :city, :state;

#### See Also

EXECUTE IMMEDIATE

PREPARE

# **EXECUTE IMMEDIATE**

The EXECUTE IMMEDIATE statement runs the embedded SQL statement contained in the specified host variable.

### Syntax

EXECUTE IMMEDIATE :stmt\_hvar

## Arguments

stmt\_hvar

Is a character string host variable that contains a Transact-SQL statement.

### Remarks

The EXECUTE IMMEDIATE statement must conform to Transact-SQL statement rules. It cannot contain input parameter markers or host variables. It cannot return results. Results returned from this statement are discarded. Additionally, the statement cannot contain keywords that pertain exclusively to ESQL/C.

If any rows are returned, SQLCODE is set to +1, which indicates an exception.

## Examples

strcpy(prep, "DELETE FROM mf\_table WHERE name='elaine'");

EXEC SQL EXECUTE IMMEDIATE :prep;

See Also

**EXECUTE** 

# FETCH

The FETCH statement retrieves a specific row from the cursor.

### Syntax

FETCH [ [ NEXT | PRIOR | FIRST | LAST ] FROM ] cursor\_name [USING DESCRIPTOR :sqlda\_struct | INTO :hvar [,...]]

## Arguments

#### NEXT

Specifies returning the first row of the result set if this FETCH statement is the first FETCH against the cursor; otherwise, specifies moving the cursor one row in the result set. NEXT is the default method used to move through a result set.

#### PRIOR

Specifies returning the previous row in the result set.

#### FIRST

Specifies moving the cursor to the first row in the result set and returning the first row.

#### LAST

Specifies moving the cursor to the last row in the result set and returning the last row.

#### cursor\_name

Is a previously declared and opened cursor.

#### sqlda\_struct

Is an output SQLDA data structure that was previously populated by the DESCRIBE statement and that contains output value addresses. This option is used only with a cursor declared by prepared SELECT statements. (SELECT statements are prepared by using the PREPARE statement.)

hvar

Is one or more host variables to receive the data.

## Remarks

If the NEXT, PRIOR, FIRST, or LAST options are not specified, the FETCH statement retrieves the next *n* rows from the result set produced by the OPEN statement for this cursor and writes the values of the columns in those rows to the corresponding host variables or to addresses specified in the SQLDA data structure.

An OPEN *cursor\_name* statement must precede a FETCH statement, and the cursor must be open while FETCH runs. Also, the data type of the host variable must be compatible with the data type of the corresponding database column.

If the number of columns is less than the number of host variables, the value of SQLWARN3 is set to W. If an error occurs, no further columns are processed. Processed columns are not undone. The SQLCODE value of 100 indicates that no more rows exist in the result set.

The USING DESCRIPTOR *:sqlda\_struct* option can be used only with a dynamically defined cursor. The INTO *:hvar* option can be used with either a dynamic or static cursor.

## Examples

```
EXEC SQL DECLARE C1 CURSOR FOR
SELECT au_fname, au_lname FROM authors FOR BROWSE;
EXEC SQL OPEN C1;
while (SQLCODE == 0)
{
EXEC SQL FETCH C1 INTO :fname, :lname;
}
```

See Also

DECLARE CURSOR

**PREPARE** 

**DESCRIBE** 

Advanced Programming

<u>OPEN</u>

# **GET CONNECTION**

The GET CONNECTION statement retrieves the DBPROCESS pointer for the specified connection and stores the pointer in a host variable for use with DB-Library function calls.

### Syntax

GET CONNECTION connection\_name INTO :hvar

## Arguments

connection\_name

Is a previously opened connection.

hvar

Is the host variable, declared as data type DBPROCESS \*. The *hvar* option is used to store the DB-Library connection pointer. The pointer can then be used with DB-Library function calls.

## Remarks

The GET CONNECTION statement stores the DB-Library DBPROCESS pointer for an ESQL/C connection in a host variable. This is useful if you want to use features or functions that are specific to DB-Library (such as **text** and **image** handling functions) in your ESQL/C program.

As with all DB-Library programs, you must first use **#define** to define the appropriate platform before you include the DB-Library header files Sqlfront.h and Sqldb.h and link to the appropriate DB-Library .lib file.

- Programs for Microsoft® Windows NT® 4.0, Microsoft Windows® 95, and Microsoft Windows 98 must first use **#define** DBNTWIN32 and then link to Ntwdblib.lib.
- Programs for 16-bit Windows and QuickWin must first use **#define**
DBMSWIN and then link to Msdblib3.lib.

• Programs for Microsoft MS-DOS<sup>®</sup> must first use **#define** DBMSDOS and then link to Msdblib3.lib.

If you are using a WHENEVER statement in your program, the Embedded SQL keyword sqlerror must not be uppercase to avoid conflict with the DB Librarydefined constant SQLERROR.

### Examples

#define DBNTWIN32
#include <windows.h>
#include <sqlfront.h>
#include <sqldb.h>

```
.
EXEC SQL BEGIN DECLARE SECTION;
DBPROCESS* dbproc;
```

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

```
EXEC SQL CONNECT TO gizmo.pubs
AS my_connection
USER sa
EXEC SQL GET CONNECTION my_connection
INTO dbproc;
if (dbproc != NULL)
{
    printf("Got DBPROCESS connection, current database is '%Fs'\n",
    dbname(dbproc));
}
```

```
else
{
    printf("ERROR: Getting DBPROCESS connection\n");
}
```

# See Also

CONNECT TO

# **OPEN**

The OPEN statement begins row-at-a-time data retrieval for a specified cursor.

### Syntax

OPEN cursor\_name [USING DESCRIPTOR :sqlda | USING :hvar [,...]]

### Arguments

cursor\_name

Is a previously declared, opened, and fetched cursor.

sqlda

Is an input SQLDA data structure that was previously constructed by the application. The SQLDA data structure contains the address, data type, and length of each input parameter. This option is used only with cursors that are declared by dynamical SQL statements.

hvar

Is one or more input host variables that correspond to parameter markers in the SELECT statement. This option is used only with cursors that are declared by dynamical SQL statements.

# Remarks

The OPEN statement runs the SELECT statement specified in the corresponding DECLARE CURSOR statement to produce a result set, which is accessed one row at a time by the FETCH statement.

If the cursor is declared with a static SELECT statement, the SELECT statement can contain host variables (*hvar*) but not parameter markers (?). Host variables can only be used in place of constants. They cannot be used in place of the names of tables, columns, other database objects, or keywords. The current values of the host variables are substituted when the OPEN statement runs. Because the OPEN statement is for a statically declared cursor, it cannot contain

the USING :hvar and USING DESCRIPTOR :sqlda options.

If the cursor is declared by using a dynamic SELECT statement, the SELECT statement can contain parameter markers but not host variables. Parameter markers can be used in place of column names in the SELECT statement. If the SELECT statement has parameter markers, the OPEN statement must include either the USING *:hvar* option with the same number of host variables, as in the SELECT statement, or the USING DESCRIPTOR *:sqlda* option that identifies the SQLDA data structure already populated by the application.

With the USING DESCRIPTOR *:sqlda* option, the values of the program variables are substituted for parameter markers in the SELECT statement. The program variables are addressed by corresponding *sqldata* entries in the SQLDA data structure. For information about SQLDA, see <u>Using the SQLDA Data</u> <u>Structure</u>.

A separate database connection is used for each open browse cursor. Each connection counts toward the total number of user connections configured on an instance of Microsoft® SQL Server<sup>™</sup> 2000. If an attempt to make a new connection fails when opening a browse cursor, or if a valid current connection is not made when opening a standard cursor, then run-time error -19521 "Open cursor failure" - usually occurs.

### Examples

```
EXEC SQL DECLARE c1 CURSOR FOR
SELECT au_fname,au_lname FROM authors FOR BROWSE;
EXEC SQL OPEN c1;
```

```
while (SQLCODE == 0)
{
    EXEC SQL FETCH c1 INTO :fname,:lname;
}
```

See Also

**CLOSE** 

SET CURSOR\_CLOSE\_ON\_COMMIT

DECLARE CURSOR

Advanced Programming

**FETCH** 

# PREPARE

The PREPARE statement prepares SQL statement from a character string in the host variable for later execution. It also associates the statement with a symbolic SQL name.

### Syntax

PREPARE stmt\_name [INTO :sqlda] FROM :hvar

# Arguments

#### stmt\_name

Is the statement to a subsequent EXECUTE or OPEN statement, or a previous DECLARE CURSOR statement.

sqlda

Is the output SQLDA data structure to be populated.

hvar

Is the host variable that contains the SQL statement.

## Remarks

The PREPARE statement processes dynamic SQL statements. Because singleton SELECT statements (SELECT INTO) are not supported in dynamic SQL statements, they are not supported in PREPARE statements.

The statement name cannot be reused in multiple PREPARE statements within the same program module (source code file). Statement names are global within a program module. PREPARE statements cannot be shared by separately compiled programs linked into a single executable module, or by a program and dynamic-link libraries (DLLs) that execute in a single process.

You can use a PREPARE statement in one of two ways:

• You can open a prepared dynamic cursor. (You must declare a dynamic

cursor by using the DECLARE CURSOR statement before you prepare a SELECT statement.)

• You can execute a prepared statement.

Prepared statement names are limited to use in a single cursor definition. The following statements used together are not valid:

### EXEC SQL DECLARE cursor1 CURSOR FOR prep\_select\_statement EXEC SQL DECLARE cursor2 CURSOR FOR prep\_select\_statement

If the PREPARE statement is used by an EXECUTE statement, *hvar* cannot contain an SQL statement that returns results.

When you use PREPARE, the SQL statement in *:hvar* cannot contain host variables or comments, but it can contain parameter markers (?). Additionally, the SQL statement cannot contain SQL keywords that pertain exclusively to Embedded SQL keywords.

The following keywords cannot be used in a PREPARE statement:

CLOSE	FETCH
COMMIT	INCLUDE
DESCRIBE	OPEN
END-EXEC	PREPARE
EXEC SQL	ROLLBACK
EXECUTE	WHENEVER

The INTO :sqlda option merges the functionality of the DESCRIBE statement with the functionality of the PREPARE statement. Therefore, the following sample statements are functionally identical:

EXEC SQL PREPARE gumbo INTO :mysqlda FROM :hamhock;

Or

EXEC SQL PREPARE gumbo FROM :hamhock;

# EXEC SQL DESCRIBE gumbo INTO :mysqlda;

Note that the SQLDA data structure is populated only for output data. Even then, the application must set the value of each *sqldata* entry in the SQLDA data structure to the address of the corresponding program variable. (The *sqldata* field is part of *sqlvar*.) The SQLDA data structure used for input parameters must be constructed entirely by the application. For more information, see <u>Using the SQLDA Data Structure</u>.

# Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char prep[] = "INSERT INTO mf_table VALUES(?,?,?)";
char name[30];
char car[30];
double num;
EXEC SQL END DECLARE SECTION;
```

EXEC SQL PREPARE prep\_stat FROM :prep;

```
while (SQLCODE == 0)
{
    strcpy(name, "Elaine");
    strcpy(car, "Lamborghini");
    num = 4.9;
    EXEC SQL EXECUTE prep_stat USING :name, :car, :num;
}
```

## See Also

DECLARE CURSOR EXECUTE

DESCRIBE

Advanced Programming

# **SELECT INTO**

The SELECT INTO statement retrieves one row of results. The SELECT INTO statement is also known as a singleton SELECT statement.

### Syntax

SELECT [select\_list] INTO {:hvar [,...]} select\_options

# Arguments

select\_list

Is the list of items (table columns or expressions) to retrieve data from.

hvar

Is one or more host variables to receive the *select\_list* items.

select\_options

Is one or more statements or other options that can be used with the Transact-SQL SELECT statement (for example, a FROM or WHERE clause). The GROUP BY, HAVING, COMPUTE, CUBE, and ROLLUP clauses are not supported.

## Remarks

The SELECT INTO statement retrieves one row of results and assigns the values of the items in *select\_list* to the host variables specified in the INTO list. If more columns are selected than the number of receiving host variables, then the value of SQLWARN3 is set to W. The data type and length of the host variable must be compatible with the value assigned to it. If data is truncated, the value of SQLWARN3 is set to W.

The Embedded SQL SELECT INTO statement is compatible with the Transact-SQL SELECT INTO statement. The Embedded SQL SELECT INTO statement is used only when results are retrieved for substitution in the application. The Transact-SQL SELECT INTO statement does not return results to the application and must be issued by using the Embedded SQL EXECUTE statement.

If more than one row is returned, SQLCODE is set to +1, which indicates an exception.

## Examples

EXEC SQL SELECT au\_lname INTO :name FROM authors WHERE

### See Also

**BEGIN DECLARE SECTION** 

END DECLARE SECTION

# SET ANSI\_DEFAULTS

The SET ANSI\_DEFAULTS statement sets ANSI defaults ON for the duration of the Microsoft® SQL Server<sup>™</sup> 2000 query-processing session or for the duration of a running trigger or a stored procedure. This statement is supported only for connections to SQL Server version 6.5 or later.

### Syntax

SET ANSI\_DEFAULTS ON;

# Arguments

ON

Specifies SQL-92 compatibility.

### Remarks

The ON option sends the Transact-SQL statement SET ANSI\_DEFAULTS ON to SQL Server and sets the following statements as shown:

SET CONCURRENCY LOCKCC

SET CURSORTYPE CUR\_STANDARD

SET CURSOR\_CLOSE\_ON\_COMMIT ON

SET FETCHBUFFER 1

SET SCROLLOPTION FORWARD

**Note** Setting the SET ANSI\_DEFAULTS statement to OFF is not supported. To reverse the effects of the SET ANSI\_DEFAULTS statement, turn off each option listed earlier.

See Also

SET CONCURRENCY

SET FETCHBUFFER SET CURSOR\_CLOSE\_ON\_COMMIT SET SCROLLOPTION SET CURSORTYPE

# SET CONCURRENCY

The SET CONCURRENCY statement sets the concurrency option for standard cursors.

### Syntax

SET CONCURRENCY {LOCKCC | OPTCC | OPTCCVAL | READONLY}

### Arguments

LOCKCC (default if SET ANSI\_DEFAULTS is ON)

Specifies intent to update locking. If a FETCH statement is issued within a user-defined transaction, an exclusive lock is placed on the data before it is fetched. The exclusive lock prevents others from viewing or changing the data until the lock is released when the transaction closes.

OPTCC (default if SET ANSI\_DEFAULTS is not ON)

Specifies optimistic concurrency control based on a **timestamp** column (if available) or all non**text**, non**image** columns.

OPTCCVAL

Specifies optimistic concurrency control based on all non**text**, non**image** columns.

READONLY

Specifies read-only cursors. Data retrieved by a FETCH statement cannot be modified.

### Remarks

After the SET CONCURRENCY statement is issued, it affects all subsequent OPEN statements. Using the DECLARE CURSOR FOR UPDATE statement has the same effect as SET CONCURRENCY LOCKCC, and any reference to the SET CONCURRENCY statement is ignored. The SET CONCURRENCY statement is also ignored if you are using browse cursors. If the LOCKCC option is used, you can choose to hold open the user-defined transaction only around each fetch. This requires that a SET FETCHBUFFER statement be issued before opening the cursor. Or you can choose to hold open the user-defined transaction for the life of the cursor. Note that holding open a transaction during LOCKCC cursor operations can significantly reduce concurrency and degrade performance.

If the OPTCC or OPTCCVAL option is used, an UPDATE WHERE CURRENT OF statement can fail if the row has been changed since the last FETCH statement. The application must be able to handle this situation.

### Examples

EXEC SQL SET CONCURRENCY READONLY;

### See Also

DECLARE CURSOR SET FETCHBUFFER SET ANSI\_DEFAULTS SET SCROLLOPTION SET CURSORTYPE Standard DB-Library Cursors

# SET CONNECTION

The SET CONNECTION statement specifies which database connection to use for subsequent SQL statements.

### Syntax

SET CONNECTION connection\_name

## Arguments

connection\_name

Is the name of an existing database connection.

### Remarks

The value for *connection\_name* must match the connection name specified in a previous CONNECT TO statement. The *connection\_name* can be either the connection's literal name or a host variable that contains character values. SET CONNECTION can be used only with a named connection.

If you are using connections across compilation modules, you must use named connections.

## Examples

EXEC SQL CONNECT TO caffe.pubs AS caffe1 USER sa; EXEC SQL CONNECT TO latte.pubs AS latte1 USER sa; EXEC SQL SET CONNECTION caffe1; EXEC SQL SELECT name FROM sysobjects INTO :name; EXEC SQL SET CONNECTION latte1; EXEC SQL SELECT name FROM sysobjects INTO :name; EXEC SQL SELECT name FROM sysobjects INTO :name; EXEC SQL DISCONNECT caffe1; EXEC SQL DISCONNECT latte1; // The first select will take place against the pubs // // database on server "caffe." The second SELECT will //
// take place against the pubs database on server "latte." //
// In place of the two "disconnect" statements at the end, //
// you can also write: //
// EXEC SQL DISCONNECT ALL; //

# See Also

CONNECT TO

# SET CURSOR\_CLOSE\_ON\_COMMIT

The SET CURSOR\_CLOSE\_ON\_COMMIT statement sets all cursors on a connection to automatically close when a COMMIT TRANSACTION or a ROLLBACK TRANSACTION statement is issued.

### Syntax

SET CURSOR\_CLOSE\_ON\_COMMIT { ON | OFF };

### Arguments

ON

Specifies closing all cursors on a connection when a COMMIT TRANSACTION or a ROLLBACK TRANSACTION statement is issued.

OFF (default)

Specifies that the calling application is required to close each cursor on a connection when a COMMIT TRANSACTION or a ROLLBACK TRANSACTION statement is issued.

### Remarks

You can use the SET CURSOR\_CLOSE\_ON\_COMMIT command for standard and browse cursors.

### See Also

<u>CLOSE</u>

SET ANSI\_DEFAULTS

<u>OPEN</u>

# SET CURSORTYPE

The SET CUSORTYPE statement sets the use of standard (DB-Library) or browse (one connection per cursor) cursors.

### Syntax

SET CURSORTYPE {CUR\_BROWSE | CUR\_STANDARD}

### Arguments

CUR\_BROWSE

Specifies using browse cursors. Each browse cursor requires a separate connection to Microsoft® SQL Server<sup>™</sup> 2000.

CUR\_STANDARD (default)

Specifies using standard DB-Library cursors. A unique index must exist on the source table. Each standard cursor shares the same (single) connection to the instance of SQL Server used by the application. This is recommended for cursors because it does not require a separate connection to SQL Server.

### Remarks

After this statement is issued, it affects all subsequent cursor OPEN statements. Using the DECLARE CURSOR FOR UPDATE statement has the same effect as SET CURSORTYPE CUR\_STANDARD.

To initiate positioned update or delete operations on a browse cursor, the SELECT statement used to open the browse cursor must include a FOR BROWSE clause, and the table must include a **timestamp** column.

The SELECT statement used to open a standard cursor cannot contain any of the following Transact-SQL clauses:

FOR BROWSE	COMPUTE
SELECT INTO	UNION

For more information about restrictions on standard cursors with a SCROLLOPTION of DYNAMIC, see <u>SET SCROLLOPTION</u>.

If a cursor definition violates any of the conditions noted earlier, an SQLCODE of -19521 is generated.

SQL Server treats each browse cursor connection as a different user. Locks held by one browse cursor can block operations attempted by other browse cursors. Because standard cursors share the same connection to SQL Server used by the application, using standard cursors eliminates potential browse cursor locking problems.

### Examples

EXEC SQL SET CURSORTYPE CUR\_STANDARD;

### See Also

DECLARE CURSOR Standard DB-Library Cursors SET CONCURRENCY Browse Cursors SET FETCHBUFFER

# SET FETCHBUFFER

The SET FETCHBUFFER statement sets internally the number of rows to be retrieved at one time for standard cursors.

### Syntax

SET FETCHBUFFER num\_rows

# Arguments

num\_rows

Is the number of rows to be retrieved at one time from Microsoft® SQL Server<sup>™</sup> 2000. Embedded SQL for C buffers *num\_rows* internally, and it returns one row to the application for each FETCH statement issued.

# Remarks

The default is 1 if SET ANSI\_DEFAULTS is ON. Otherwise, the default is 10. After this statement is issued, it affects all subsequent cursor OPEN statements. Because a FETCH statement returns only a single row, the SET FETCHBUFFER statement is for performance tuning only.

The number of fetch operations required to retrieve all data from a cursor can be reduced by setting the *num\_rows* parameter higher than the default setting of 10. You should ensure that the client has enough available memory for you to increase this setting. You might need to reduce the default setting of the *num\_rows* parameter if client memory is constrained, or if the cursors have many columns or wide columns.

# Examples

EXEC SQL SET FETCHBUFFER 5;

See Also

SET CONCURRENCY SET SCROLLOPTION SET CURSORTYPE Standard DB-Library Cursors

# **SET OPTION**

The SET OPTION statement sets values for query-processing options for Microsoft® SQL Server<sup>™</sup> 2000.

### Syntax

SET OPTION {QUERYTIME | LOGINTIME | APPLICATION | HOST} value

# Arguments

### QUERYTIME

Specifies the number of seconds that DB-Library waits for SQL Server to respond to a Transact-SQL statement. The default value is 0 seconds, meaning forever. This option is functionally the same as the DB-Library **dbsettime** function.

### LOGINTIME

Specifies the number of seconds that DB-Library waits for SQL Server to respond to a request for a DBPROCESS connection. The default value is 10 seconds. This option is functionally the same as the DB-Library **dbsetlogintime** function.

#### APPLICATION

Specifies the application name in the LOGINREQ structure. This option supports a character value only (not numerical). This option is functionally the same as the DB-Library **DBSETLAPP** function.

#### HOST

Specifies the workstation name in the LOGINREQ structure. This option supports a character value only (not numerical). This option is functionally the same as the DB-Library **DBSETLHOST** function.

#### value

Is the valid numerical time-out value, in seconds, for the QUERYTIME or LOGINTIME DB-Library option. It is also the valid character value for the

APPLICATION or HOST DB-Library option. *value* can be a character literal or a host variable.

## Remarks

The remaining DB-Library options either do not apply to Embedded SQL (for example, row buffering) or can be set by using Transact-SQL statements.

# Examples

EXEC SQL SET OPTION LOGINTIME 5; EXEC SQL CONNECT TO caffe.pubs USER sa; // If login to the server "caffe" does not occur within five // // seconds, the "connect" attempt times out and SQL Server returns to t EXEC SQL SET OPTION QUERYTIME 2; EXEC SQL SELECT name FROM sysobjects INTO :name; // If the query response does not occur within two seconds, // // the query attempt times out and returns to the program. //

## See Also

**Using Host Variables** 

# SET SCROLLOPTION

The SET SCROLLOPTION statement sets the scrolling functionality and row membership for standard cursors.

### Syntax

SET SCROLLOPTION {DYNAMIC | FORWARD | KEYSET}

# Arguments

DYNAMIC

Specifies that row membership in the cursor is updated for every FETCH statement, and that the cursor scrolls forward and backward.

FORWARD (default)

Specifies that row membership in the cursor is updated for every FETCH statement, and that the cursor scrolls only forward.

KEYSET

Specifies fixing row membership in the cursor at OPEN time.

# Remarks

After this statement is issued, it affects all subsequent cursor OPEN statements. This statement is ignored if you are using browse cursors.

Row membership is determined by the subset of rows defined by the SELECT statement criteria and available for retrieval by using FETCH statements. This includes the specific data rows available and what (if any) order those rows are in.

For KEYSET cursors, row membership is fixed when the OPEN statement is issued. After a KEYSET cursor has been opened, if additional rows that meet the SELECT statement criteria are added by another user, they are not visible within the opened cursor. You must close and reopen a KEYSET cursor to see the added rows. For DYNAMIC and FORWARD cursors, row membership is dynamic. Each FETCH statement retrieves the most current data available. After a DYNAMIC or FORWARD cursor has been opened, if rows that meet the SELECT statement criteria are added by another user, they are potentially visible within the opened cursor.

When using the DYNAMIC or FORWARD scroll options, the SELECT statement used to open the cursor cannot contain a GROUP BY or HAVING clause or an ORDER BY clause without a unique index.

### Examples

EXEC SQL SET SCROLLOPTION DYNAMIC;

See Also

SET CONCURRENCY

SET FETCHBUFFER

SET CURSORTYPE

Standard DB-Library Cursors
# UPDATE

The UPDATE statement changes data in the row where the cursor is currently positioned.

## Syntax

UPDATE {*table\_name* | *view\_name*} SET {*column=expression*[,...]} WHERE CURRENT OF *cursor\_name* 

## Arguments

table\_name

Is the table to be updated.

view\_name

Is the view to be updated.

column

Is the column to be updated.

expression

Is the value of a particular column name. This value can be an expression or a null value.

cursor\_name

Is a previously declared, opened, and fetched cursor.

## Remarks

In addition to having the searched update functionality of the Transact-SQL UPDATE statement, the Embedded SQL UPDATE statement includes functionality that is known as positioned update. Positioned update changes the row most recently fetched by a cursor.

Note In a positioned update, the WHERE CURRENT OF option is used in place

of a search condition clause. The WHERE CURRENT OF option cannot be used in a PREPARE statement.

In a positioned update that uses a browse cursor, the SELECT statement used to open the cursor must include a FOR BROWSE clause, and the base table(s) must include a **timestamp** column. If an error prevents any row found by the search condition WHERE CURRENT OF from being deleted, no changes are made to the database.

When using a browse cursor, or a standard cursor with optimistic concurrency control (SET CONCURRENCY with the OPTCC or OPTCCVAL option), and the row has been changed after the last FETCH statement, no changes are made to the database. The value of SQLCODE is set to -532, which means that a positioned UPDATE or DELETE statement failed because of a conflict with another user. Also, the SQLERRD3 field in the SQLCA data structure shows no rows processed.

# Examples

```
while (SQLCODE == 0)
{
 EXEC SQL FETCH c1 INTO :fname,:lname;
 if (SQLCODE == 0)
 {
   printf("%s %s", fname, lname);
   printf("Update? ");
   scanf("%c", &reply);
   if (reply == 'y')
   ł
     printf("New last name? ");
     scanf("%s", &lname);
     EXEC SQL
      UPDATE authors SET au lname=:lname
      WHERE CURRENT OF c1:
     printf("update sqlcode= %s", SQLCODE);
   }
```

} }

See Also

DECLARE CURSOR FETCH

# **UPDATE (Searched)**

The UPDATE (Searched) statement changes data in existing rows of a table. UPDATE (Searched) is a standard Transact-SQL statement.

## Syntax

```
UPDATE {table_name | view_name} SET [table_name. | view_name.]
{column_name={expression | NULL | (select_statement)}[,...]} [FROM
{table_name | view_name}[,...]] [WHERE search_condition]
```

## Arguments

table\_name

Is the table to be updated.

view\_name

Is the view to be updated.

column\_name

Is the column to be updated.

expression

Is the value of a particular column. This value must be an expression.

select\_statement

Is a valid SELECT statement that returns a single row with a single column of data.

search\_condition

Is any expression that can legally follow the standard Transact-SQL WHERE clause.

## Remarks

Use UPDATE to change values. Use INSERT to add new rows.

Updating a **varchar** or **text** column with the empty string (' ') inserts a single space. All **char** columns are padded to the defined length.

All trailing spaces are removed from **varchar** column data. Strings that contain only spaces are truncated to a single space.

The SQL batch size of 128 KB limits the maximum amount of data that you can alter with UPDATE. Because some memory is required for the query's execution plan, the actual amount of data you can include in an UPDATE statement is somewhat less than 128 KB. For example, you can update one column of about 125 KB, or two columns of about 60 KB each.

## Examples

UPDATE authors SET au\_fname = 'Fred' WHERE au\_lname = 'White'

### See Also

**INSERT** 

# WHENEVER

The WHENEVER statement specifies the action (CONTINUE, GOTO, or CALL) to be taken when one of three possible SQLCODE conditions is met following the execution of an Embedded SQL statement.

### **Syntax**

WHENEVER {SQLWARNING | SQLERROR | NOT FOUND} {CONTINUE | GOTO *stmt\_label* | CALL *function()*}

## Arguments

### SQLWARNING

Specifies that an Embedded SQL warning occurred and was stored in the SQLCA data structure.

### SQLERROR

Specifies that a Microsoft® SQL Server<sup>™</sup> 2000 message was received and stored in the SQLCA data structure.

#### NOT FOUND

Specifies that no rows were returned from a valid and properly executed SELECT statement, or that a FETCH statement returned no more rows, and that SQLCODE was set to 100 in the SQLCA data structure.

#### CONTINUE (default)

Specifies running the next physically sequential statement in the source program.

#### stmt\_label

Is the place in the program where control is assumed.

#### function()

Is a function in your application. Parentheses () are required following the function name (*function*()). If parentheses are omitted, the function is not

called. The function can include parameters.

## Remarks

SQLCODE conditions have the following values.

Condition	Value	Example
No error	0	
NOT FOUND	100	Fetch past end of results
SQLWARNING	+1	Data truncation on output
SQLERROR	< 0 (negative)	Constraint violation

For more information about how to handle specific error conditions programmatically, using SQLSTATE values, see <u>SQLSTATE Messages</u>.

The following SQLCODE values are revised in ESQL/C version 6.5 and are carried to version 7.0.

Condition	New value	Previous value
Singleton SELECT statement returns	-1	1
more than 1 row		
NULL value returns, but no indicator	-1	0
variable declared		
Second attempt to open cursor without	-1	0
corresponding close while		
CLOSE_ON_COMMIT is in force		
Server error encountered on cursor open	-nnn, where	-19521
	<i>nnn</i> is a server	
	message	
	number	

WHENEVER statement actions are related to the position of statements in the source code, not in the run sequence. The default is CONTINUE for all conditions.

## Examples

EXEC SQL WHENEVER sqlerror GOTO displayca;

EXEC SQL WHENEVER sqlerror CALL error\_funct(param);

# See Also

•

Using the WHENEVER Statement

Advanced Programming

# **Building Applications**

To build an Embedded SQL for C (ESQL/C) application for the Microsoft® Windows NT® 4.0, Microsoft Windows® 95, or Microsoft Windows 98 operating systems, you must compile the application on a computer running Windows NT 4.0, Windows 95, or Windows 98 on the Intel® platform.

If you need to build an ESQL/C application for either 16-bit Windows or Microsoft MS-DOS®, you must use the ESQL/C compilation environment in Microsoft SQL Server<sup>™</sup> 6.0 or 6.5. SQL Server 7.0 does not supply the 16-bit binaries needed to compile ESQL/C clients in these environments:

- To build an ESQL/C application for the 16-bit Windows operating system, you must compile the application on a computer running MS-DOS or 16-bit Windows.
- To build an ESQL/C application for the MS-DOS operating system, you must compile the application on a computer running MS-DOS version 6.22 or later.

The following illustration shows how application code that contains Embedded SQL statements is precompiled, compiled, and linked, and how the application operates at run time.

7]

When you compile an ESQL/C application, the general process of creating an executable program is the same regardless of the operating system you compile for. The specific procedures for precompiling, compiling, and linking ESQL/C applications for Windows NT 4.0, Windows 95, or Windows 98, 16-bit Windows, and MS-DOS are described in other topics.

# **Steps for Building an Application**

The following steps apply to building all Embedded SQL for C (ESQL/C) applications:

1. Run the appropriate **nsqlprep** precompiler to prepare the ESQL/C program for compiling with a C compiler.

For more information about **nsqlprep**, see <u>Running the nsqlprep</u> <u>Precompiler</u>.

- 2. Compile the C program created by the precompiler by using an appropriate C compiler. The compiler creates object file(s).
- 3. Link the object file(s), ESQL/C library files, and any other needed library files to create an executable file for the desired environment.

For more information about compiling and linking ESQL/C applications, see the ESQL/C samples in the C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\Esqlc subdirectory.

# **Embedded SQL Applications at Run Time**

When you run an Embedded SQL for C (ESQL/C) application, SQL statements are executed as follows:

- 1. For every SQL statement, the application calls the appropriate ESQL/C run-time services.
- 2. If the SQL statement is static, the run-time services execute the SQL statement or a previously compiled stored procedure with the appropriate input parameters. Executing a stored procedure at run-time depends on your pre-compile options to **nsqlprep** (that is, /SQLACCESS which creates stored procedures for static SQL statements). If the statement is dynamic, the run-time services issue the SQL statements directly.
- 3. The run-time services use DB-Library calls to send and retrieve data to and from Microsoft® SQL Server<sup>™</sup> 2000.
- 4. The run-time services insert data into C program host variables (or SQLDA data structures). Status and error information is inserted into the SQLCA data structure.

# **Processing Embedded SQL Statements**

Embedded SQL for C (ESQL/C) fully supports all Transact-SQL extensions, including stored procedures, local variables, and control-of-flow language. Due to syntax conflicts with Embedded SQL reserved keywords, and because static SQL statements are compiled into stored procedures by the precompiler, minor restrictions apply to how Transact-SQL extensions are implemented. The restrictions are as follows:

- The Transact-SQL EXECUTE statement should be abbreviated to EXEC to avoid conflict with the Embedded SQL EXECUTE statement.
- Transact-SQL statement labels should not be used in static SQL statements because they conflict with the syntax for host variables. However, you can use Transact-SQL labels in dynamic SQL statements.

If Microsoft<sup>®</sup> SQL Server<sup>™</sup> 2000 returns an error during compilation, the SQL Server error code appears in the precompiler error message as a negative number, for example:

```
SQL Syntax Error -SQLCODE = -207
```

For more information about a list of messages returned by ESQL, see <u>Embedded SQL for C Messages</u>.

• Because **nsqlprep** converts all static SQL statements into stored procedures, all limitations for Transact-SQL stored procedures apply to static SQL statements. A static SQL statement that contains a single transaction-management statement (such as COMMIT TRANSACTION or SAVEPOINT) is not compiled into a stored procedure. Instead, the statement is issued dynamically at run time.

For more information about restrictions that relate to other Embedded SQL statements, see <u>Embedded SQL Statements</u>.

# **Access Plans and Bind Files**

The static SQL statements of an application can be issued at run time (as are dynamic SQL statements), or they can be placed into an access plan. An access plan is a set of stored procedures. It includes a separate stored procedure for each static SQL statement.

Using the **nsqlprep** precompiler, you can connect to a specified server and database and create an access plan. Each time an application is recompiled, old stored procedures with the same program module name are dropped.

If the database you need to connect to is unavailable when you are ready to compile, you can use the **nsqlprep** precompiler to create a bind file. A bind file is a Transact-SQL script used to create stored procedures for the access plan. You can later use the Microsoft® SQL Server<sup>TM</sup> 2000 **isql** utility to apply the bind file to the database before you run the application.

# **Running the nsqlprep Precompiler**

You must run the appropriate **nsqlprep** precompiler to prepare your Embedded SQL for C (ESQL/C) program for compiling with a C compiler.

Operating system	Precompiler
Microsoft <sup>®</sup> Windows NT <sup>®</sup> 4.0 (Intel <sup>®</sup> platform)	Nsqlprep.exe
Microsoft Windows® 95 or Windows 98	Nsqlprep.exe
16-bit Windows	Sqlprep.exe
Microsoft MS-DOS®	Sqlprep.exe

The **nsqlprep** precompiler finds SQL statements, parses the SQL statements, and if applicable, creates an access plan or bind file. The precompiler creates a C program that can be compiled with an appropriate C compiler. For more information about compiling and linking, see <u>Compiling and Linking Embedded</u> <u>SQL Applications</u>.

# **Setting Up the nsqlprep Precompiler**

Before you run the **nsqlprep** precompiler, do the following:

The **nsqlprep** precompiler uses your compiler to process header files

- Set the INCLUDE environment variable to include the full path where the Sqlca.h and Sqlda.h ESQL/C header files are located and set the LIB environment variable to include the full path where the library files are located. There are several ways to accomplish this, including either:
  - Issuing a SET statement at the command prompt, such as

```
SET INCLUDE = C\Mssql17\DevTools\INCLUSE;
%include%
```

SET LIB = C\Mssql17\DevTools\LIB; %LIB%

 At the command prompt, first running Vcvar32.bat (in the \Program Files\Microsoft Visual Studio\VC98\Bin directory) and then running setenv.bat (in \Mssql17\DevTools\Samples\Esqlc directory).

The **nsqlprep** precompiler automatically includes these header files in the C programs it creates. Do not explicitly include them (by using **#include**) in an Embedded SQL program.

• To enable communication with Microsoft® SQL Server<sup>™</sup> 2000, ensure that an appropriate Net-Library is loaded or available on the path when precompiling with the **/DB** and **/PASS** options. For example, the Named Pipes Net-Library for the Microsoft Windows NT® operating system (Intel® platform) is Dbnmpntw.dll, and the Named Pipes Net-Library for the Microsoft MS-DOS® operating system is the Dbnmpipe.exe TSR.

# **Precompiler Syntax**

You can use either a slash (/) or a hyphen (-) to designate an **nsqlprep** precompiler option. For example, /**DB** *server\_name.database\_name* and -**DB** *server\_name.database\_name* are equivalent.

### **Syntax**

nsqlprep program\_file\_name [/SQLACCESS | /NOSQLACCESS]
[/FLAGGER {ENTRY | NONE}] [/DB [server\_name.]database\_name
/PASS {login[.password] | \$INTEGRATED}] [/BIND file\_name]
[/MSG file\_name] [/NOLOGO] [/PLAN name] [/NOLINES]
[/user\_defined\_option]

## Arguments

### program\_file\_name

Is the file name (without the extension) of the Embedded SQL program to precompile. The precompiler searches for the file name and the file extension .sqc. For example, if you run **nsqlprep** myprogrm, **nsqlprep** searches for Myprogrm.sqc and precompiles it if it is found.

### /SQLACCESS

Specifies that **nsqlprep** will create stored procedures automatically for the static SQL statements in the program. You must also include **/DB** and **/PASS** to specify a direct connection to Microsoft® SQL Server<sup>™</sup> 2000 for creating the stored procedures, or **/BIND** to create a bind file for later loading of the stored procedures. Note that for standard cursors, the original SELECT statement is used directly, and the stored procedures created by the **/SQLACCESS** option are not used.

### /NOSQLACCESS

Specifies that **nsqlprep** will not automatically create stored procedures for static SQL statements in the program. If you also include **/DB** and **/PASS** to specify a direct connection to SQL Server, **nsqlprep**:

- Issues a message.
- Connects to SQL Server.
- Drops stored procedures created by a previous precompile of the same program.
- Completes the precompile without creating new stored procedures.

#### /FLAGGER

Specifies sending static SQL statements to the server at compile time for syntax checking. Syntax or compilation error messages generated from the server are displayed on the screen. If you specify the **/FLAGGER** option, you must also specify the **/DB** and **/PASS** options. You cannot use the **/PLAN** or **/SQLACCESS** options with the **/FLAGGER** option.

#### ENTRY

Specifies that static SQL statements are also checked for FIPS 127-2 SQL compliance. If the precompiler encounters SQL statements that do not conform to the specified FIPS level of support, an error message is displayed on the screen or stored in the file specified by the /MSG option. Program output is generated the same way as when you use the /NOSQLACCESS option (that is, stored procedures are not created).

#### NONE

Specifies that static SQL statements are not checked for FIPS 127-2 SQL compliance but are checked for correct syntax. The existence of database objects that the statements refer to is also verified.

#### /DB [server\_name.]database\_name

Specifies a database, and optionally a server running SQL Server, in which to put stored procedures for the access plan. If SQL Server is running on your local computer, you need to supply only the *database\_name*. Whenever you use the **/DB** option, you must also use the **/PASS** option. *server\_name* or *database\_name* is the same server or database name in the Embedded SQL

CONNECT TO statement in the program.. For more information about using /**DB**, see <u>Access Plan and Bind File Options</u>.

/PASS login[.password]

Specifies the user identifier and password for SQL Server access and stored procedure creation. Whenever you use the **/PASS** option, you must also use the **/DB** option. *login* and *password* are a user's login ID and password, or a user's login ID only. For more information about using **/PASS**, see <u>Access</u> <u>Plan and Bind File Options</u>.

### **\$INTEGRATED**

Forces use of Windows Authentication support for the *login*[.*password*] parameter. If Windows Authentication support is forced, any implicit runtime connection also uses Windows Authentication.

### /BIND file\_name

Causes creation of a bind file on precompiler execution. The extension .bnd is appended to the required *file\_name* supplied. *file\_name* is an MS-DOS path and file name specification. The file name should not include an extension.

The bind file is an **isql** script that can be used to create the desired stored procedures (it includes CREATE PROCEDURE statements). For more information about using **/BIND**, see <u>Access Plan and Bind File Options</u>.

When you create a bind file, you must use the Embedded SQL CONNECT TO statement to connect to the server. If you use this option and no stored procedure is created, the bind file is deleted.

### /MSG file\_name

Causes creation of a text file containing warning and error messages generated by **nsqlprep** processing. The .msg extension is appended to the specified file name. If **nsqlprep** processing does not generate any warning or error messages, the file is not generated. *file\_name* is a Microsoft MS-DOS® path and file name specification. The file name should not include an extension.

### /NOLOGO

Specifies suppression of the **nsqlprep** banner and the compiler banner (**nsqlprep** invokes the compiler).

#### /PLAN name

Specifies a nondefault name for an access plan. (The default name is the *program\_file\_name* without an .sql extension.)

### /NOLINES

Specifies that the generated .c file be displayed for debugging, instead of the .sqc file. You must use the **/NOLINES** option if the .sqc source code contains C language code that appears on the same line after an EXEC SQL statement.

### */user\_defined\_option*

Is a user-defined option to be passed to the C compiler.

# **Access Plan and Bind File Options**

By using the **/DB** and **/PASS** options with the **nsqlprep** precompiler, you can connect to a specified server and database and create an access plan (set of stored procedures) for each separately compiled program module (compilation unit). With the **/DB** and **/PASS** options, **nsqlprep** makes a connection using the specified server name, database name, login ID, and password. The access plan consists of a separate stored procedure for each static SQL statement in each compiled program module.

By default, the stored procedure names consist of the following:

- Name of the program module
- A date/timestamp converted to eight printable ASCII characters
- A dollar sign (\$)
- The access plan section number

The date/timestamp provides for re-creating stored procedures for the access plan with identical program module names each time the program is compiled. However, if you use the **/PLAN** option and specify a nondefault plan name that ends with an underscore (\_), **nsqlprep** does not include the date/timestamp in the stored procedure names. In that case, you can reuse stored procedures created from a previous precompile.

If the database you need to connect to is unavailable or you do not want to use it when you are ready to compile, you can use the **/BIND** option in the **nsqlprep** precompiler command line to create a bind file. The bind file is a Transact-SQL script used to create stored procedures for access plans. You can later use the Microsoft® SQL Server<sup>™</sup> 2000 **isql** utility to apply the bind file to the database. However, to run a bind file as an SQL script, each line must contain no more than 1,000 characters if you want to use the **isql** utility. If one or more lines contain more than 1,000 characters, use the **isqlw** utility (SQL Query Analyzer). You must apply the bind file to the database before you can run the

corresponding C application.

If you do not specify the **/DB** and **/PASS** options or the **/BIND** option, both static and dynamic SQL statements are issued at run time.

# **Compiling and Linking Embedded SQL Applications**

The **nsqlprep** precompiler creates a C program with the file extension .c from an Embedded SQL for C (ESQL/C) program. For example, from the ESQL/C program Myprogrm.sqc, **nsqlprep** creates a C program named Myprogrm.c. You can compile the C program for the Intel® platform by using the appropriate C compiler.

The ESQL/C precompiler does not support preprocessing of C++ modules. To use Embedded SQL in a C++ application, you must create separate C-language modules for data access functions and preprocess only those modules. ESQL/C also does not support the use of precompiled headers because the first step in preprocessing is to expand all #include files. You can improve compilation time by segregating data access code into .sqc modules that have a minimum number of #include files.

If your program contains a #include windows.h statement, you must precede it with the following two statements:

#define \_OLE2\_H\_
#define NOIME

In this example, the Ole2.h and Imm.h header files are excluded. The Ole2.h and Imm.h header files are not compatible with version 6.5 of ESQL/C.

The Transact-SQL keyword *null* should not be uppercase in Embedded SQL programs to avoid conflict with the C keyword NULL. Also, the Embedded SQL keyword *delete* and the Transact-SQL keyword *in* should not be uppercase in applications for 32-bit Windows to avoid conflict with 32-bit Microsoft Windows®-defined constants in Windows.h.
# **Compiling and Linking for Windows NT and Windows 95 or Windows 98**

You can compile and link precompiled ESQL/C programs for Microsoft® Windows NT® and Microsoft Windows® 95 or Microsoft Windows 98 on a computer running the Windows NT 4.0, Windows 95, or Windows 98 operating system by using Microsoft Visual C++® version 4.5 or later.

You can set the LIB environment variable to avoid specifying library paths for the Embedded SQL libraries, for example:

SET LIB= C:\Program Files\Microsoft SQL Server\80\Tools\DevTools

The following libraries are supplied and used by ESQL/C programs when built for Windows NT 4.0 and Windows 95 or Windows 98:

- Caw32.lib
- Sqlakw32.lib

# To compile and link a program for Windows NT 4.0, Windows 95 or Windows 98

 Run the compiler Cl.exe as you would for Windows NT 4.0 or Windows 95 or Windows 98 to create an object file, for example: CL /c /W3 /D"\_X86\_" MYPROGRM.C

In this example, the compiler creates the object file Myprogrm.obj. The /**D**"\_**X86**\_" compiler option defines the symbol necessary for Windows NT 4.0, Windows 95, or Windows 98 operating systems that run on Intel®-based computers.

You can use any additional compiler options allowed by the compiler.

2. Run the linker (Link.exe) as you would for Windows NT 4.0, Windows 95 or Windows 98 to link the compiled ESQL/C object file and system libraries, which creates an executable file for Windows NT 4.0, Windows 95, or Windows 98. When linking files for Windows NT 4.0, Windows 95, or Windows 98 programs, you must explicitly link with the Sqlakw32.lib and Caw32.lib libraries; for example: LINK /NOD /subsystem:windows MYPROGRM.OBJ MYPR GDI32.LIB USER32.LIB LIBCMT.LIB SQLAKW32.LIB C/

In the example, the compiled object file Myprogrm.obj, system libraries, and the ESQL/C libraries Sqlakw32.lib and Caw32.lib are linked together to create the executable file Myprogrm.exe for Windows NT 4.0, Windows 95, or Windows 98.

To run a Windows NT 4.0, Windows 95, or Windows 98 ESQL/C application, the dynamic-link libraries Sqlakw32.dll, Ntwdblib.dll, and Dbnmpntw.dll (or other appropriate Net-Library) must be available in the path.

# **Project Settings for Visual C++**

The following project settings enable application development in Microsoft® Visual C++®. The techniques discussed apply to any integrated development environment. For more information about a specific manufacturer's compiler and integrated development environment, see the compiler documentation.

You set up ESQL/C projects in the same way you set up projects that include any additional component libraries.

#### To add the needed directory names to Visual C++ environment settings

- 1. On the **Tools** menu, click **Options**.
- 2. Click the **Directories** tab.
- 3. In the **Show directories for** box, click **Include files**.
- 4. Enter the path for Microsoft® SQL Server<sup>™</sup> 2000 development include files, typically x:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Include.
- 5. In the **Show directories for** box, click **Library files**.
- 6. Enter the path for SQL Server development library files, typically x:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Include.
- 7. In the **Show directories for** box, click **Executable files**.
- 8. Enter the path for SQL Server development binary executable files, typically C:\Program Files\Microsoft SQL Server\80\Tools\Binn.

You must change the executable directories list so Visual C++ will correctly

locate the **nsqlprep** executable file.

The following illustration shows adding the path for header files to the Visual C++ include file path list.

Options		? ×
Editor   Tabs   Debug   Compat	ibility Build Directories	3
Platform:	Show directories for:	
Win32	Include files	-
Directories:	🖱 🗙 -	<del>+ +</del>
c:\mssql7\include		
c:\odbcsdk\include		
C:\MsDev\VC\INCLUDE		
C:\MsDev\VC\MFC\include		
C:\MsDev\VC\ATL\include		
C:\MsDev\SharedIDE\bin\%include%	<u> </u>	
	OK	Cancel

As you create each ESQL/C source file, you must indicate the preprocessor steps necessary to generate the appropriate C source for compilation. Depending on the version of Visual C++, use the **Build** menu or the **Project** menu to locate the compilation settings for the project.

Compilation settings can be set at the source-file level, and the custom build steps for your ESQL/C source files are set at this level as well.

### To add custom build steps for an ESQL/C source file

- 1. On either the **Project** menu (Visual C++ 5.0) or the **Build** menu (Visual C++ 4.*x*), click **Settings**.
- 2. Click the ESQL/C source file in the project's file list. In Visual C++ 5.0 a single instance of the file appears in the project's list. You can click **All configurations** in the **Settings for** box to set custom build commands for both debug and release versions of your project. In Visual C++ 4.*x*, you can expand the file list and select each occurrence of the ESQL/C source file to set the custom build commands for all project configurations.

- 3. Click the **Custom Build** tab.
- 4. In the **Build commands** box, enter the **nsqlprep** command. You can use the \$(InputName) macro to specify the file.
- 5. In the **Output file(s)** box, specify the name of the output file. The output file is a C source file. You can use the \$(InputName) macro to specify the file.

The illustration below shows custom build settings for an ESQL/C source file.

Project Settings	? ×
Settings For: Win32 Debug	General       Custom Build         Input file:       .\genwin.sqc         Description:       Running ESQL/C precompiler         Build command(s):       Image: Image
<u> </u>	Directory  Files  Dependencies
	OK Cancel

After you've successfully preprocessed the ESQL/C source files, you can add the generated C source to your project.

# **Compiling and Linking for 16-bit Windows**

Microsoft® SQL Server<sup>™</sup> 2000 does not supply the environment for compiling ESQL/C 16-bit Microsoft Windows® clients (the 16-bit binaries for Windows are not included). However, these clients will run under SQL Server 2000. To compile an ESQL/C 16-bit Windows client, you need to use the ESQL/C compilation environment in SQL Server 6.0 or 6.5, which includes the 16-bit Windows binaries.

You can compile and link precompiled ESQL/C programs for 16-bit Windows on a computer running the Microsoft MS-DOS® or 16-bit Windows operating system by using the Microsoft Visual C++® development system (16-bit), version 1.52 or later.

ESQL/C source files can be built as either large or medium memory model 16bit Windows executable files.

You can set the LIB environment variable to avoid specifying library paths for the ESQL/C libraries, for example:

SET LIB= C:\Program Files\Microsoft SQL Server\80\Tools\DevTools

The following libraries are supplied and used by ESQL/C programs when built for 16-bit Windows:

- Caw.lib
- Sqlakw.lib

### To compile and link a program for 16-bit Windows

 Run the compiler Cl.exe as you would for 16-bit Windows to create an object file, for example: CL /c /W3 /AL MYPROGRM.C

In the example, the compiler creates the object file Myprogrm.obj. The /AL compiler option sets the memory model of the program to large. You can use any additional compiler options allowed by the compiler.

 Run the linker Link.exe as you would for 16-bit Windows to link the compiled ESQL/C object file and system libraries, which creates an executable file for Windows. When linking files for Windows-based programs, you must explicitly link with the Sqlakw.lib and Caw.lib libraries, for example: LINK /NOD MYPROGRM.OBJ,,, LIBW.LIB LLIBCEW.LIE MYPROGRM.DEF;

In the example, the compiled object file Myprogrm.obj, system libraries Libw.lib and Mlibcew.lib, and ESQL/C libraries Sqlakw.lib and Caw.lib are linked together to create the large model, executable file Myprogrm.exe for Windows.

To run a 16-bit Windows-based ESQL/C application, the dynamic-link libraries Sqlakw.dll, Msdblib3.dll, and Dbnmp3.dll (or other appropriate Net-Library) must be available in your path.

# **Compiling and Linking for MS-DOS**

Microsoft® SQL Server<sup>™</sup> 2000 does not supply the environment for compiling ESQL/C 16-bit Microsoft MS-DOS® clients (the 16-bit binaries for MS-DOS are not included). However, these clients will run under SQL Server 2000. To compile an ESQL/C 16-bit MS-DOS client, you need to use the ESQL/C compilation environment in SQL Server 6.0 or 6.5, which includes the 16-bit MS-DOS binaries.

You can compile and link precompiled ESQL/C programs for MS-DOS on a computer running the MS-DOS version 6.22 or later operating system by using the Microsoft Visual C++® development system (16-bit), version 1.52 or later.

ESQL/C sources can be built as large memory model MS-DOS executable files.

You can set the LIB environment variable to avoid specifying library paths for the ESQL/C libraries, for example:

SET LIB= C:\Program Files\Microsoft SQL Server\80\Tools\DevTools

The following libraries are supplied and used by ESQL/C programs when built for MS-DOS:

- Car.lib
- Rldblib.lib
- Sqlakd.lib

### To compile and link a program for MS-DOS

 Run the compiler Cl.exe as you would for MS-DOS to create an object file, for example: CL /c /W3 /AL MYPROGRM.C

In the example, the compiler creates the object file Myprogrm.obj. The /AL compiler option sets the memory model of the program to large.

You can use any additional compiler options allowed by the compiler.

 Run the linker Link.exe as you would for MS-DOS to link the compiled ESQL/C object file and system libraries, which creates an executable file for MS-DOS. When linking files for MS-DOS-based programs, you must explicitly link with the Sqlakd.lib, Rldblib.lib, Oldnames.lib and Car.lib libraries, for example: LINK /NOD MYPROGRM.OBJ,,, LLIBCE.LIB SQLAKD.L RLDBLIB.LIB, MYPROGRM.DEF;

In the example, the compiled object file Myprogrm.obj, system library Llibce.lib, ESQL/C libraries Sqlakd.lib, Car.lib, and Oldnames.lib and the DB-Library library Rldblib.lib are linked together to create the executable file Myprogrm.exe for MS-DOS.

To run an MS-DOS-based ESQL/C application, load the appropriate Net-Library TSR.

# **Compiling and Linking for QuickWin**

QuickWin is a set of libraries that helps convert source code for Microsoft® MS-DOS® into 16-bit Microsoft Windows®-based applications. You can compile and link precompiled ESQL/C programs for QuickWin on a computer that is running the MS-DOS or 16-bit Windows operating system by using the Microsoft Visual C++® development system (16-bit), version 1.52.

Microsoft® SQL Server<sup>™</sup> version 7.0 does not supply the environment for compiling ESQL/C 16-bit MS-DOS or 16-bit Windows clients (the 16-bit binaries for Windows and MS-DOS are not included). However, these clients will run under SQL Server 7.0. To compile an ESQL/C 16-bit MS-DOS or 16-bit Windows client, you need to use the ESQL/C compilation environment in SQL Server 6.0 or 6.5, which includes the 16-bit binaries.

ESQL/C sources can be built as either large or medium memory model 16-bit Windows, QuickWin executables.

You can set the LIB environment variable to avoid specifying library paths for the ESQL/C libraries, for example:

SET LIB= C:\Program Files\Microsoft SQL Server\80\Tools\DevTools

The following libraries are supplied and used by ESQL/C for C programs when built as 16-bit Windows, QuickWin executable files:

- Caw.lib
- Sqlakw.lib

### To compile and link a program for QuickWin

 Run the compiler Cl.exe as you would for 16-bit Windows to create an object file, for example: CL /c /W3 /AM MYPROGRM.C

In the example, the compiler creates the object file Myprogrm.obj. The /AM compiler option sets the memory model of the program to

medium. You can use any additional compiler options allowed by the compiler.

2. Run the linker Link.exe as you would for 16-bit Windows to link the compiled ESQL/C object file and system libraries, which creates an executable file for Windows. When linking files for Windows-based programs, you must explicitly link with the Sqlakw.lib and Caw.lib libraries, for example: LINK /NOD MYPROGRM.OBJ,,, LIBW.LIB MLIBCEWQ.I

MYPROGRM.DEF;

In the example, the compiled object file Myprogrm.obj, system libraries Libw.lib and Mlibcewq.lib, and ESQL/C libraries Sqlakw.lib and Caw.lib are linked together to create the executable file Myprogrm.exe for Windows.

To run a 16-bit Windows-based ESQL/C application, the dynamic-link libraries Sqlakw.dll, Msdblib3.dll, and Dbnmp3.dll (or other appropriate Net-Library) must be available in your path.

# **Debugging Embedded SQL Programs**

You can use the Microsoft debuggers (including the integrated debugger in 16bit Microsoft® Visual C++® and 32-bit Visual C++), Microsoft CodeView® window-oriented debugger for Microsoft Windows®, and WinDebug for Microsoft Windows NT® 4.0, Windows 95, and Windows 98 to examine your Embedded SQL for C (ESQL/C) program while it is executing. A Microsoft debugger displays ESQL/C source statements as lines of executable code. While debugging, you can set breakpoints on Embedded SQL statements to test host variables. To monitor SQLCA and SQLDA fields, you must monitor the SQLCA and SQLDA data structures and member variables. For example, you must monitor sqlca->sqlcode instead of the SQLCODE macro, and you must monitor sqlca->sqlerrd[0] instead of the SQLERRD1 macro.

When you use the integrated debugger in 16-bit Visual C++ or 32-bit Visual C++, you can set breakpoints on lines of source code in your .sqc file. However, to begin a debugging session, you must first open the .c file generated by **nsqlprep** and make that window active. Then on the **Debug** menu, click **Go**.

If you are using Visual C++ 5.0 for example, you can open the .exe file, and choose from the menu BUILD, START DEBUG, STEP INTO. For this to work, you need to compile and link with a debug information switches. For example:

## **Precompile:**

nsqlprep myprogram /NOLINES /NOLOGO /NOSQLACCESS

## **Compile:**

cl -c -G4d -W3 -Zi -Od myprogram.c

## Link:

Link /MAP /DEBUG.full /DEBUGTYPE:both /SUBSYSTEM:console

For debugging the .c source code, you need to supply the /NOLINES precompiler option to **nsqlprep**. This allows debugging directly into the .c file instead of the .sqc file.

When you use CodeView for 16-bit Windows and step through the code, the cursor disappears until it reaches the next C statement.

## See Also

Precompiler Syntax

# **Advanced Programming**

Most of the features used in writing Embedded SQL code are discussed in Embedded SQL Programming.

Advanced programming topics include:

- Data type mappings from Transact-SQL to C, and vice versa
- The SQLDA data structure for data input and output
- The contents of the SQLCA data structure
- The EXEC statement and how to use it to selectively bypass the creation of an access plan

Embedded SQL for C is not thread-safe. If you are using ESQL/C in a threaded application, use only ESQL/C calls from a single thread of execution. It is best if you use the main thread.

If you place the ESQL/C calls in a thread other than the main thread, the thread can be started only one time. The thread must then remain available for all subsequent ESQL/C needs. Therefore, if you place the ESQL/C calls in a thread, and then start and stop that thread multiple times to accomplish database tasks, you may experience unexpected behavior. To implement a thread that handles all ESQL/C activity, set up the thread with an event-triggering mechanism.

In ESQL/C, statements that return only one result set can be executed. For example:

CREATE PROCEDURE spTest AS SELECT au\_lname FROM authors SELECT au\_fname FROM authors

## GO

Execution of **spTest** stored procedure produces two result sets from Microsoft® SQL Server<sup>™</sup> 2000, but ESQL/C application will see only the first result set.

## See Also

Embedded SQL Programming

# **Data Type Mapping for Embedded SQL**

Embedded SQL for C (ESQL/C) maps C data types to Microsoft® SQL Server<sup>™</sup> 2000 data types, and vice versa. No Unicode data types are supported in ESQL/C (for example, the data types **nvarchar**, **nchar**, and **ntext** are not supported). Conversions are supported for all non-Unicode data types except **datetime** or **smalldatetime**, **money** or **smallmoney**, and, in some instances, **decimal** or **numeric**. Data types that can be converted to **datetime** or **smalldatetime**, **money** or **smallmoney**, and **decimal** or **numeric** are indicated by T.

	Assigned SQL	datetime or	money or	decimal or
C data type	Server data type	smalldatetime	smallmoney	numeric
short	smallint	F	F	F
int	smallint	F	F	F
long	int	F	F	F
float	real	F	F	F
double	float	F	F	F
char	varchar[x] (1)	Т	Т	Т
void *p	binary (2)	Т	Т	Т
char byte	tinyint	F	F	F

1 For more information, see <u>Mapping Character Data Types</u>.

2 For more information, see <u>Pointers as Host Variables</u>.

The above table is valid for SQL Server version 6.5. In SQL Server 7.0, mapping from one type to another is done by the database server. So all the above conversions are true in SQL Server 7.0.

Output data is truncated if the receiving data type is too short. This causes an exception (warning), and the SQLCODE field of the SQLCA data structure is set to +1. Input data can be truncated if the receiving SQL Server column is too short. However, in this case, no exception is generated.

Because **text** data types are not allowed in stored procedures, you cannot use C fields that are more than 255 bytes long in static SQL statements compiled into access plans.

# **Mapping Character Data Types**

You can declare character data types as host variables with a fixed maximum length by explicitly supplying the length of the character array, or by declaring the host variable with an initial value, for example:

```
EXEC SQL BEGIN DECLARE SECTION;
char var1[18];
char var2[] = "Initialized string";
EXEC SQL END DECLARE SECTION;
```

In the example, two host variables are declared, each with a maximum length of 18 bytes.

Host variables declared as single-byte characters with no explicit length or initial values (for example, **char var**[3]) are treated as 1-byte integer data types.

Here is how C **character** data types are mapped to or from the Microsoft® SQL Server<sup>™</sup> 2000 **char**, **varchar**, or **text** data types.

Mapped data types	When converted
From C <b>character</b> to SQL	Data is copied and truncated or padded
Server <b>char, varchar</b> , or <b>text</b>	with blanks if the SQL Server table
	receiving the columns is set to a fixed
	length.
To C <b>character</b> from SQL	Data is copied and, if necessary, truncated
Server <b>char, varchar</b> , or <b>text</b>	to the length of the receiving field and is
	terminated with a NULL. If data is
	truncated, the SQLWARN1 field of the
	SQLCA data structure is set.

## **Pointers as Host Variables**

A C host variable, declared as a pointer to a void (*void* \**p*), is treated as a data buffer of unknown length. Your program must verify that enough memory is allocated to hold any output data received. You may also need to modify the generated .c code manually to specify the length of the variable. For example, if you declare the *void* \**pChar* variable, the precompiler generates a statement like the following.

Sqlasetv(2, 0, 462, (short)-1, (void far \*)pChar, (void far \*)0, (void far

The third parameter of -1 indicates that this is a pointer. You may have to modify this length to indicate proper size of data contained in *pChar*. The .c source can then be compiled and linked as usual.

## Mapping date or time Data

Because C does not have a **date** or **time** data type, Microsoft® SQL Server<sup>TM</sup> 2000 **date** or **time** columns are converted to characters by using the SQL Server default date format, for example:

#### *mm dd yyyy hh:mm:ss*[**am** | **pm**]

You can send dates to SQL Server from C character fields by using any of the character date formats accepted by SQL Server. For a date without a time, use an 11-byte receiving field. If data is truncated, the SQLWARN3 field of the SQLCA data structure is set. For other formats, use the Transact-SQL CONVERT statement. For example, to convert a time without a date, use a statement similar to this:

## SELECT CONVERT(char, date, 8) FROM sales

When you attempt to put a fixed-length data type into a buffer that is too small for it, an error occurs and no data is copied.

# **Mapping Binary Data**

On singleton SELECT statements and cursor fetches, Microsoft® SQL Server<sup>™</sup> 2000 **binary**, **varbinary**, and **image** columns are mapped to C host variables declared as character arrays without any data conversion. The host variable is treated as a byte array. To retrieve the hexadecimal character representation of a **binary**, **varbinary**, or **image** column, use the Transact-SQL CONVERT function on the column.

When input, a C **character array** data type can be mapped to a SQL Server **binary**, **varbinary**, or **image** column by using dynamic SQL statements. To do this, use two question marks (??) as parameter markers. The data format on input is the raw binary data (not its hexadecimal character representation.) If you include at least one space between the parameter marker and its indicator, you can use indicator variables. You cannot use static SQL statements to map binary data on input parameters.

You can use static SQL statements to map binary data on input parameters. However, you will need to manually modify the generated .c code to specify the length of the variable. For example, if you declare the *void* \**pChar* variable, the precompiler will generate a sqlasetv to set the parameter length to unknown by putting -1 as the parameter length. The third parameter of -1 indicates that this is a pointer. You may have to modify this length to indicate proper size of data contained in *pChar*. The .c source can then be compiled and linked as usual.

# **Preparing SQLDA for Data Input and Output**

The SQLDA data structure contains descriptive information about each input parameter or output column. The structure contains the column name, data type, length, and a pointer to the actual data buffer for each input or output parameter.

For output data that uses the SQLDA data structure, you can use the DESCRIBE statement (or the PREPARE statement with the INTO option) to enter the column name, data type, and other data into the appropriate fields of the SQLDA data structure. DESCRIBE also sets *sqld* to the number of dynamic host variables used in the SQL statement being described.

Before using the SQLDA data structure in a PREPARE INTO or DESCRIBE statement, your application must set the size of the SQLDA data structure in bytes and the maximum number of entries. These numbers are reflected in the *sqldabc* and *sqln* fields, respectively.

Before performing a FETCH statement, the application must insert into the *sqldata* field the address of each program variable that will receive the data from the corresponding column. (The *sqldata* field is part of *sqlvar* within the SQLDA data structure.) If indicator variables are used, *sqlind* must also be set to the corresponding address. The data type field (*sqltype*) and length field (*sqllen*) are filled with information in the Microsoft® SQL Server<sup>TM</sup> 2000 column from a PREPARE INTO or a DESCRIBE statement. The value in the data type and length fields can be overridden by the application before a FETCH statement is executed. The *sqltype* code and the *sqld* address that are assigned must be one of the valid data types listed in Valid Values for *sqltype*. For more information, see <u>Valid Values for sqltype</u>.

Before the DESCRIBE or PREPARE statement is issued, the value of *sqln* must be set higher than the anticipated number of output columns. If the number of columns is unknown, you can set *sqln* to 0, and then issue a DESCRIBE statement. No column detail information is moved into the SQLDA data structure, but the number of columns in the result set is inserted into *sqld*.

To use the SQLDA data structure for input data, your application must supply data for the fields of the entire SQLDA data structure, including the *sqln*, *sqld*, *sqldabc*, *sqltype*, *sqllen*, and *sqldata* fields for each variable. If the *sqltype* field

has an odd code number (value), the address of the indicator variable must also be supplied.

# SQLDA Data Structure

The SQLDA data structure definition (from Sqlda.h) looks like this:

// SQL Descriptor Area - SQLDA				
struct sqlda				
{				
	unsigned char sqldaid[8];	// Eye catcher = 'SQLDA '		
	long sqldabc;	// SQLDA size in bytes = 16+44*SQLN		
	short sqln;	// Number of SQLVAR elements		
	short sqld;	// Num of used SQLVAR elements		
	struct sqlvar			
	{			
	short sqltype;	// Variable data type		
	short sqllen;	// Variable data length		
		// Maximum amount of data < 32K		
	unsigned char FAR			
	*sqldata;	// Pointer to variable data value		
	short FAR *sqlind;	// Pointer to null indicator		
----	--------------------	------------------------------		
	struct sqlname	// Variable name		
	{			
	short length;	// Name length [130]		
	unsigned char			
	data[30];	// Variable or column name		
	} sqlname;			
	} sqlvar[1];			
};				

## SQLDA

Here are fields and data types for the SQLDA data structure.

Field	Data type	Contains	
sqldaid	unsigned char*	Text string SQLDA. This field is not used for FETCH, OPEN, or EXECUTE statements.	
sqldabc	long	Length of the SQLDA data structure ( $sqln^*$ 44 + 16).	
sqln	short	Total number of <i>sqlvar</i> entries allocated. Equal to the number of input parameters or output columns.	
sqld	short	Number of <i>sqlvar</i> entries used.	
sqlvar	struct	Values listed in the <i>sqlvar</i> table. The values can occur several times, listed once per column in the result set or input parameter.	
sqltype	short	Number that represents the data type of columns or host variables, and that indicates whether null values are allowed. For information about valid values, see <u>Valid</u> <u>Values for sqltype</u> .	
sqllen	short	External length of a value from a column.	
sqldata	unsigned char far*	Address of the host variable (which must be inserted by the application) for FETCH, OPEN, and EXECUTE statements. For DESCRIBE and PREPARE statements, <i>sqldata</i> is not used.	
sqlind short far*		Address of an indicator variable for FETCH, OPEN, and EXECUTE statements, if one exists.	
		If the column does not permit a null value, the field is undefined. If the column permits	

		a null value, <i>sqlind</i> is set to -1 if the data	
		value is null, or to 0 if the data value is not	
		null. For DESCRIBE and PREPARE	
		statements, <i>sqlind</i> is not used.	
sqlname	struct	Name and length of the column (not used for	
		FETCH, OPEN, and EXECUTE statements).	
length	short	Length of the <i>name</i> column.	
name	unsigned char*	Name of the column. For a derived column,	
		this field contains the ASCII numeric literal	
		value that represents the derived column's	
		original position within the select list.	

## Valid Values for sqltype

Here are the values for the *sqltype* field in the SQLDA data structure and corresponding Microsoft® SQL Server<sup>™</sup> 2000 data types for which they can serve as host variables in a FETCH statement or an EXECUTE statement. For each pair of *sqltype* codes, the odd number type signifies a host variable with a corresponding null indicator variable needed for setting or retrieving null values.

<i>sqltype</i> code	Description	SQL Server data type	Sample declaration
392/393	26-byte date and time <b>char</b> format corresponds to the formats supported by <b>dbconvert</b> for <b>datetime</b> to/from <b>char</b> <sup>1</sup> .	datetime, smalldatetime	char date1[27] = Mar 7 1988 7:12PM;
444/445	Binary	binary, varbinary, image, timestamp	char binary1[4097];
		<b>Note</b> <i>sqltype</i> 444/445 is automatically used for these SQL Server column types on output.	
452/453	<b>Char</b> string <=254 bytes. Not automatically null- terminated.	<b>char, varchar</b> , or <b>text</b>	char mychar[255];
	<b>Note</b> Make sure you initialize the full array with nulls when using this type for output.		

456/457	Length-prefixed long character field. Not automatically null- terminated.	<b>char, varchar,</b> or <b>text</b>	<pre>struct TEXTVAR {     short len;     char data[4097]; } textvar;</pre>
462/463	Null-terminated string. <b>Note</b> Declarations of known length ( <i>mychar1</i> ) are padded with blanks and a terminating null. Declarations of <b>char</b> pointers ( <i>mychar2</i> ) are not padded with blanks and the application must ensure sufficient space is allocated.	<b>char, varchar,</b> or <b>text</b>	char mychar1[41]; char * mychar2;
480/481	8-byte floating point.	float, real, int, smallint, tinyint, decimal, numeric, money, smallmoney	double mydouble1;
482/483	4-byte floating point.	float, real, int, smallint, tinyint, decimal, numeric, money, smallmoney	float myfloat1;
496/497	4-byte integer.	int, smallint, tinyint, bit	long myint1;
500/501	2-byte integer.	smallint, tinyint, bit	short myshort1;

<sup>1</sup> For more information about **datetime** conversion, see <u>dbconvert</u>.

Here are the data type codes returned when using SQLDA structures in

#### DESCRIBE or PREPARE INTO statements.

	<i>sqltype</i> returned by DESCRIBE	
	Or DREDARE	
SQL Server column	INTO	Comments
char, varchar	452/453	452/453 is a COBOL <b>char</b>
		data type. Not null-terminated.
		It is easier to use 462/463
		from C.
text	456/457	<i>sqllen</i> set to maximum of
		32767 for <b>text</b> .
binary, varbinary,	444/445	<i>sqllen</i> set to maximum of
image, timestamp		32767 for <b>image</b> .
smallint, tinyint, bit	500/501	
int	496/497	
float	480/481	
real	482/483	
datetime, smalldatetime	392/393	
decimal, numeric,	484/485	COBOL decimal format. Not
money, smallmoney		supported for FETCH or
		EXECUTE in ESQL/C (use
		480 or 482 instead). <i>Sqllen</i>
		encoded with scale and
		precision. Use <i>sqllen</i> &= 0xFF
		to get just precision.

DESCRIBE and PREPARE INTO statements are only supported for output columns of SELECT statements. They are not supported for INSERT, UPDATE, or DELETE statements, or for any statement requiring an input host variable.

## SQLCA Data Structure

The SQLCA data structure contains status information about the Embedded SQL statement last executed. The structure definition for SQLCA (from Sqlca.h) looks like this:

// SQL C	ommunication Area - SQLCA	
typedef s	truct sqlca	
{		
	unsigned char	
	sqlcaid[EYECATCH_LEN];	// Eyecatcher = 'SQLCA '
	long sqlcabc;	// SQLCA size in bytes = 13
	long sqlcode;	// SQL return code
	short sqlerrml;	// Length for SQLERRMC
	unsigned char	
	sqlerrmc[SQLERRMC_SIZ];	// Error message tokens
	unsigned char sqlerrp[8];	// Diagnostic information
	long sqlerrd[6];	// Diagnostic information
	unsigned char sqlwarn[8];	// Warning flags

	unsigned char sqlext[3];	// Reserved
	unsigned char sqlstate[5];	// new member
} SQLCA;		

# SQLCA

Here are fields and data types for the SQLCA data structure.

Field	C data type	Contains	
sqlcaid	unsigned char	Text string SQLCA.	
sqlabc	long	Length of the SQLCA data structure.	
sqlcode	long	<ul> <li>Status code for the last-run SQL statement:</li> <li>0 The statement ran without error.</li> <li>1 The statement ran, but an exception was generated.</li> <li>100 A FETCH statement was issued, but no more rows satisfy the SELECT statement criteria used to define the cursor. No rows were processed.</li> <li>&lt; 0 (negative) The statement did not run due to an application, database, system, or network error.</li> </ul>	
sqlerrm		Error messages that consist of two parts.	
	sqlerrml	Length of the error message in <i>sqlerrmc</i> (0 to 70).	
	short		
	sqlerrmc	Text of the error message. Error messages longer than 70 bytes are truncated.	
	unsigned char	Reserved (diagnostic information).	
sqlerrp	unsigned char		
sqlerrd	long	Array of six integer status codes (codes not in the fields listed later are reserved).	
	sqlerrd[1]	Microsoft <sup>®</sup> SQL Server <sup>™</sup> 2000 error number.	
	sqlerrd[2]	SQL Server severity level.	
	sqlerrd[3]	Number of rows affected.	
	sqlwarn	Eight warning flags, each containing a blank or	

		W (flags not in the fields listed later are	
		reserved).	
sqlwarn[0]	unsigned char	Summary of all warning fields. Blank indicates	
		no warnings.	
sqlwarn[1]	<i>lwarn[1]</i> <b>unsigned char</b> W indicates a character string was truncated		
		during output binding	
sqlwarn[2]	unsigned char	Not used.	
sqlwarn[3]	<i>qlwarn[3]</i> <b>unsigned char</b> W indicates that the number of columns doe		
		not match the number of host variables.	
sqlext	unsigned char	Reserved.	
sqlstate	unsigned char	SQLSTATE run-time error codes.	

**WARNING** If the number of host variables and parameter markers does not match, SQLWARN3 is set to W. This condition is considered an exception (SQLCODE is set to +1). Exceeding the number of host variables or SQLDA data structure entries is fatal (SQLCODE = -19313). During a FETCH statement or a singleton SELECT statement, SQLWARN3 is set if the number of columns is not equal to the number of host variables (or SQLDA data structure entries). The lower of the two is the number of items actually processed. For more information about SQLCA, see <u>Using the SQLCA Data Structure</u>.

#### **Selectively Bypassing the Creation of Access Plans**

To run a stored procedure by using a static SQL statement, include an EXEC statement before the stored procedure call to bypass the creation of an access plan. If you do not include an EXECUTE statement before the stored procedure call, a stored procedure is created to run the stored procedure that was called. Using EXECUTE eliminates this extra step; for example:

EXEC SQL EXEC sp-addlogin :loginame, :password;

You can also use the selective bypass method to issue an SQL statement dynamically from a static SQL statement without using the PREPARE and EXECUTE statements, for example:

EXEC SQL EXEC CREATE TABLE t1 (c1 int, ...);

The current values of any host variables are substituted into the statement.

## **Defining Cursors**

Standard cursors can be used to scroll through and update a result set. Standard cursors do not require a unique index for Microsoft® SQL Server<sup>TM</sup> version 6.5 and later, do not require a timestamp or a second connection to a database for updates, and do not create a copy of the entire result set.

**Note** Standard cursors will be enhanced in later versions; however, browse cursors will not.

Behavior	Standard	Browse
Connections to the server	All share the same connection.	One per cursor, plus one base connection for all updates.
timestamp column	Not required.	Required for positioned update or delete operations.
Transaction behavior	All cursors share the same transaction space. Standard cursors cannot block each other.	Read-only cursors can block updates made by the same program.
Unique index	Not required for SQL Server version 6.5 and later.	Required for update only.
Data currency	Data values are always current. Row membership depends on the SCROLLOPTION setting: • If keyset-driven, membership is fixed at cursor- open time.	For read-only cursors, all data is current. For updatable cursors, the server takes a snapshot of the data and stores it in the <b>tempdb</b> database when the cursor is opened.

These are the cursor types and their behavior:

	• If dynamic, membership changes.	
Restrictions on use of SQL language	Cannot be used on stored procedures. Cannot contain UNION, GROUP BY, and HAVING clauses. In SQL Server version 6.5 and later, dynamic cursors can contain an ORDER BY clause if the table has a unique index.	Updatable cursors require the FOR BROWSE clause.

#### **Standard DB-Library Cursors**

Embedded SQL for C (ESQL/C) programs use the SET SCROLLOPTION, SET CONCURRENCY, SET FETCHBUFFER, and SET CURSOR\_CLOSE\_ON\_COMMIT statements to control cursor options. All other standard cursor behavior is handled automatically by the ESQL/C library.

Standard cursors use the current database connection. Because standard cursors do not use separate database connections (as browse cursors do), many of the locking problems experienced with the use of browse cursors are avoided. Standard cursors also use the cursor functions of DB-Library and will take advantage automatically of any future performance enhancements made to DB-Library cursors.

#### **Cursors and Stored Procedures**

sp_databases	sp_indexes
sp_tables	sp_fkeys
sp_columns	sp_table_privileges
sp_stored_procedures	sp_column_privileges
sp_sproc_columns	sp_server_info
sp_pkeys	

Microsoft cursors use catalog stored procedures, which include:

#### **Cursor Sensitivity to Change**

In ESQL/C, standard cursors fall into one of two categories, depending on their sensitivity to change:

Keyset-Driven

Values can change, but order and membership in the result set remain fixed at cursor-open time.

Dynamic

Values, order, and membership in the result set can change.

## **Keyset-driven Standard Cursors**

In a keyset-driven cursor, the membership and order of rows in the result set are fixed at cursor-open time, but value changes made by the cursor owner and committed changes made by other users are visible. If a change disqualifies a row for membership, or affects the order of a row, the row does not disappear or move unless the cursor is closed and reopened. Inserted phantom rows data does not appear, but changes to existing data do appear as the rows are fetched.

Specify keyset-driven cursors by issuing the SET SCROLLOPTION KEYSET statement. Dynamic standard cursors are the default if no SET SCROLLOPTION statement is issued.

In a keyset-driven cursor, all keys for the result set are kept locally (which is one reason a unique index is required). Given the results of *n* rows, the keyset contains the same *n* rows in the result set. The fetch buffer contains 10 rows by default, and moves forward through the keysets as each FETCH statement is executed. If the SET ANSI\_DEFAULTS ON statement is issued, the fetch buffer contains 1 row. You can modify the size of the fetch buffer by using the SET FETCH\_BUFFER statement.

Although values can change between fetches, rows do not move around if the changes affect ORDER BY columns, and they do not disappear if they no longer satisfy the WHERE clause.

## **Dynamic Standard Cursors**

In a dynamic standard cursor, committed changes made by anyone and uncommitted changes made by the cursor owner become visible the next time the user fetches data. Changes include insertions and deletions, as well as changes in order and membership. (Deleted rows do not leave holes.) Dynamic standard cursors cannot use a GROUP BY or HAVING clause. Dynamic standard cursors can use an ORDER BY clause in Microsoft® SQL Server<sup>TM</sup> version 6.5 and later, but only if the table has a unique index.

Specify dynamic standard cursors by issuing the SET SCROLLOPTION DYNAMIC statement. This statement sets the keyset equal to the size of the fetch buffer, which is one row in Embedded SQL. Given a result set of *n* rows, the keyset and the fetch buffer are identical. They both contain a single row, which moves forward through the result set as each FETCH statement is executed.

## **Concurrency Control**

Standard cursors control, through several options, concurrent access. With concurrent access, data soon becomes unreliable without some type of control. To activate the particular concurrency control desired, specify one of these options in a SET CONCURRENCY statement.

Option	Result
READONLY	Updates are not permitted.
LOCKCC	Rows are locked when they are fetched inside a user- initiated transaction. No other user can update these rows. Updates issued by the cursor owner are guaranteed to succeed.
	With Microsoft® SQL Server <sup>™</sup> 2000, locks placed by LOCKCC prevent other users from reading and updating the locked data. Use the BEGIN TRANSACTION and COMMIT TRANSACTION statements to hold the locks. For more information about locking, see <u>Holding Locks</u> .
OPTCC and	Fetched rows are not locked. Other users can update
OPTCCVAL	or read them.

To detect collisions between updates issued by the cursor owner and those issued by other users, standard cursors save and compare timestamps or column values. Therefore, if you specify either of the optimistic concurrency control options (OPTCC or OPTCCVAL), you may want to design the application to retry updates that fail because of collisions with other updates.

The two optimistic concurrency control options differ in the way they detect collisions.

Option	Method of detection
OPTCC	Compares timestamps if available; otherwise, saves
	and then compares the value of all non <b>text</b> , non <b>image</b>
	columns in the tables with their previous values.

OPTCCVAL	Compares all non <b>text</b> , non <b>image</b> values whether a
	timestamp is available.

## **Holding Locks**

When using standard cursors with Microsoft® SQL Server<sup>™</sup> 2000, the duration of locks acquired during cursor operations is controlled by the application. In other words, an application that uses SET CONCURRENCY LOCKCC must also issue a BEGIN TRANSACTION statement for the locking to have any effect. To hold the lock on the currently fetched row when LOCKCC is used, the application must issue a BEGIN TRANSACTION statement before each FETCH statement and a COMMIT TRANSACTION statement after all operations on that row are complete.

For repeatable-read consistency, specify HOLDLOCK in the SELECT statement when opening the standard cursor, and issue a BEGIN TRANSACTION statement before the first FETCH statement. Locks are obtained as the data is fetched and are retained until the application issues a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement.

#### **Browse Cursors**

Browse cursors are implemented as separate database connections and are treated as separate users, which can cause locking conflicts between cursors and UPDATE statements.

### **UPDATE and DELETE Statements**

Positioned UPDATE and DELETE statements are used in conjunction with browse cursors and include WHERE CURRENT OF clauses instead of search condition clauses. The WHERE CURRENT OF clause specifies the location of the corresponding cursor.

Before a cursor can be used by a positioned UPDATE or DELETE statement, the SELECT statement in the cursor declaration must contain the FOR BROWSE option. (The Microsoft® SQL Server<sup>™</sup> 2000 FOR BROWSE option is similar to the FOR UPDATE option in other SQL databases, but you must use SQL Server syntax.) To use the FOR BROWSE option, the table must have both a unique index and a **timestamp** column.

When performing a positioned UPDATE or DELETE statement, a method called optimistic concurrency control helps prevent conflicts with other users. Optimistic concurrency control allows users to share data with less interference than they would experience with locking, which is the alternative concurrency control method.

Although optimistic concurrency control minimizes the likelihood of conflicts with other users, write your application so that it can handle updates to tables that are rejected due to locking conflicts or other problems. Use the SQLCODE field in the SQL communications area (SQLCA) data structure to detect conflicts with other users. (A SQLCODE value of -532 means the positioned UPDATE or DELETE statement failed because of a conflict with another user.) For more information about the SQLCODE field, see <u>Using the SQLCA Data Structure</u>. For more information about browse-mode processing, see <u>Advanced Programming</u>.

A positioned update can be performed twice on the same row. To do this, use the FETCH statement to obtain the row, begin a transaction by using BEGIN TRANSACTION, and update a nonkey column to itself. This locks the row and prevents other users from reading or updating it until a COMMIT TRANSACTION statement is issued.

#### **Isolation Levels**

An isolation level determines the degree to which data is isolated for use by one process and guarded against interference from other processes. With browse cursors, isolation level is controlled on a per-cursor basis in Microsoft® SQL Server<sup>TM</sup> 2000.

If you do not specify the HOLDLOCK option in the DECLARE CURSOR statement, the isolation level is similar to cursor stability. SQL Server maintains only a share lock on a single row of the database as you retrieve rows with the FETCH statement by using a browse cursor. As long as the cursor is located on a given row, no other process can update that data page.

When you use the HOLDLOCK option, the isolation level is set to *repeatable read*. With the FETCH statement, and by using a browse cursor, SQL Server maintains a share lock on each fetched page of the database. No updates are permitted to the fetched data of the result set as long as the cursor is open, no matter what its position in the table. The repeatable read isolation level is useful when you want to scan a result set and produce a self-consistent summary report without locking the entire result set. Other users can update rows of the result set that have not been fetched, but fetched rows cannot be updated until the cursor is closed. When a cursor is declared with HOLDLOCK, the lock is freed when the cursor closes. To reread a result set without freeing the lock, reopen the cursor without closing it.

When you use a DECLARE CURSOR statement with the FOR BROWSE option (which is required for UPDATE or DELETE WHERE CURRENT OF statements), SQL Server makes a snapshot of the result set when the cursor is opened. No locks are placed on the original data; the cursor cannot detect any changes that are made to the data as it fetches rows. If the cursor is reopened, SQL Server makes a new snapshot of the data, so the results may not be the same.

You can use a SELECT with HOLDLOCK to place share locks on a set of rows that prevents other connections from updating the rows, and then open a FOR BROWSE cursor to repeatedly read the rows. If any modifications are needed, an UPDATE with a WHERE CURRENT OF clause can be used to update the row at the cursor position. Because the SELECT with HOLDLOCK was issued by the same connection, the share locks acquired by the HOLDLOCK do not block the UPDATE statements. All the locks are freed when the transaction is committed or rolled back.

The following example shows the use of the HOLDLOCK option with a browse cursor:

```
/* Declare a cursor for browse. */
EXEC SQL DECLARE CURSOR c1 FOR SELECT * FROM orders F
```

```
/* Begin a transaction using dynamic SQL. */
strcpy(prep, "begin transaction");
EXEC SQL EXECUTE IMMEDIATE :prep;
```

/\* Issue a singleton SELECT statement that checks all rows but return one row of output only. \*/ EXEC SQL SELECT COUNT(\*) INTO :count FROM orders HOLDL

/\* The result set is now locked until the transaction is complete. Open the cursor previously declared for browse, do some fetches and updates, close it, reopen it, and so on. \*/ EXEC SQL OPEN c1;

```
while (SQLCODE ==0)
{
  EXEC SQL FETCH c1 INTO :order_struct;
```

```
EXEC SQL UPDATE orders SET trancode = :new_code
WHERE CURRENT OF c1;
}
```

EXEC SQL CLOSE c1; EXEC SQL OPEN c1; /\* Some fetch and update operations can be done here, and the tables will not be changed. \*/

EXEC SQL CLOSE c2; strcpy(prep, "commit transaction"); EXEC SQL EXECUTE IMMEDIATE :prep; /\* Now all locks are free. \*/

#### **Cursors and Lock Conflicts**

Updates are issued through a single database connection, including updates that reference a cursor (for example, in an UPDATE WHERE CURRENT OF *cursor\_name* statement). Because of this, locking conflicts do not occur between updates issued under the same CONNECT TO, SET CONNECT, or default connection.

Each cursor's retrieval operations are performed through a separate database connection.

These situations block cursors:

• Cursors declared without FOR BROWSE place a read lock on the current row as they move through the result set. No changes can be made to that row by anyone, not even the cursor owner, until the cursor moves on.

When the cursor attempts to read a row that contains an uncommitted change (made by anyone, including the cursor owner), the cursor waits until the change is committed.

• Cursors declared with FOR BROWSE must wait for uncommitted changes (made by anyone, including the cursor owner) only during the OPEN CURSOR operation. After the cursor is open, subsequent changes do not cause the cursor to wait. When a cursor is reopened, it can be blocked by uncommitted changes.

**Note** Microsoft® SQL Server<sup>™</sup> version 6.5 performs locking internally at the page level rather than at the row level. Therefore, a second operation can be locked out by the first cursor operation even though the operations are accessing different rows.

For information about SQL Server cursor locking mechanisms, see <u>Cursor</u> <u>Transaction Isolation Levels</u>.
# **Reserved Keywords**

Here are the reserved keywords for Embedded SQL for C (ESQL/C). Keywords reserved for Transact-SQL are also reserved for ESQL/C. The application can use these keywords in string literals in SQL statements if the literal is enclosed in single quotes.

APPLICATION	FETCH	OPTCCVAL
CALL	FETCHBUFFER	OPTION
CLOSE	FOUND	QUERYTIME
CONCURRENCY	GET	READONLY
CONNECT	HOST	SCROLLOPTION
CONNECTION	IMMEDIATE	SECTION
CUR_BROWSE	INCLUDE	SQLCA
CUR_STANDARD	INDICATOR	SQLDA
CURRENT	KEYSET	SQLERROR
CURSOR	LOCKCC	SQLWARNING
CURSORTYPE	LOGINTIME	USER
DESCRIBE	MIXED	USING
DESCRIPTOR	NOT	WHENEVER
DISCONNECT	OF	WORK
DYNAMIC	OPEN	
FORWARD	OPTCC	

# **Embedded SQL for C Messages**

Because Embedded SQL for C (ESQL/C) messages are not generated by Microsoft® SQL Server<sup>TM</sup> 2000, they do not appear in the **sysmessages** table.

Message	Run time/	
number	compile time	Description
-4998	С	Attempt to connect to the specified database server failed.
-19031	С	Unable to open bindfile.
-19051	С	Too many sections.
-19101	R	Statement too long.
-19103	R	Illegal %s value %s.
		Nonnumeric %s value %s.
		(Invalid number for the time-out value.)
-19104	R/C	Incorrect SQL statement syntax.
-19199	С	ESQL keyword(s) detected in PREPARE statement.
-19306	С	Host variable used but not declared.
-19313	R	Too few host variables.
-19324	С	Host variable may not be used in this context.
-19408	R	Invalid SQL data type for SQL_TYP_DECIMAL.
-19413	R	Data overflow occurred during decimal data
		conversion.
-19422	R	Unknown SQL Server data type.
-19423	R	Invalid destination data type.
-19501	R	No cursor declared.
-19505	С	Duplicate cursor name: %s.
-19508	R	Cursor not positioned on a row.
-19514	R	Cursor not prepared.
-19517	R	Cursor open attempted for non-SELECT prepared statement.
-19521	R	Open cursor failure for section % <i>d</i> of plans.

-19523	R	Failure to locate/close cursor. Section % <i>d</i> , plan % <i>s</i> .
-19524	R	Table for this cursor not updatable.
-19525	R	Attempt to fetch on unopened cursor.
-19526	R	No access plan for this cursor.
-19527	R	Could not get section for this cursor.
-19528	R	Connection for section <i>%d</i> of plan <i>%s</i> has NULL <b>DBPROCESS</b> .
-19701	R	NULL connection name.
		Connection % <i>s</i> not found.
-19702	R	Connection name not found.
		Attempt to close nonexistent connection.
-19703	R	Failed to get <b>DBPROCESS</b> .
		Autoconnect failure.
-19706	R	Login failure in section % <i>d</i> .
-19707	R	Duplicate connection name.
-19822	R	Improperly initialized user SQLDA.
-19911	С	The SQL data type specified for a host variable is invalid.
-19913	С	The token identifier has already been used.
-19917	С	Invalid or incorrect option to <b>sqlainit()</b> .
-19946	С	Cursor % <i>s</i> not declared.
-19953	С	Invalid call type.
-19955	R	Text not found in <i>%s</i> section <i>%u</i> .
-19956	R	Access plan section or statement text not found.
-19957	R	Access plan or statement text not found.
-19994	R	Cannot run next BEGIN DECLARE sections. Statement ignored.
-19995	R	END DECLARE encountered without preceding BEGIN DECLARE statement. Statement ignored.
-19999	С	An internal error occurred.

Error Message Severity Levels

Embedded SQL for C Error Message Format

# **SQLSTATE Messages**

SQLSTATE codes return values that are error, warning, or "no data found" messages. Here's how SQLSTATE and SQLCODE messages correspond to each other.

SQLSTATE code	SQLCODE code
First digit greater than 0	Negative number
First digit equal to 1	1
First digit equal to 2	100

If a WHENEVER SQLERROR CALL *myproc()* statement is in effect, *myproc()* is called when SQLSTATE returns a value with a nonzero first digit.

Use SQLSTATE to check for exceptions and warnings. To use SQLSTATE in a program, you must explicitly declare SQLSTATE within a host variable declaration block, for example:

#### EXEC SQL BEGIN DECLARE SECTION;

Errors caught at compile time by using the **/SQLACCESS** option will not be mapped to SQLSTATE values at run time.

Microsoft<sup>®</sup> SQL Server<sup>™</sup> 2000 errors map to these SQLSTATE codes.

	SQL Server message	
	or	
SQLSTATE	DB-Library error	
code	numbers	Description
01001	532,10095 (DB-	Cursor update or delete failure due to
	Library)	optimistic concurrency check failure.
01003	8153	Elimination of null values by set
		operator (warning).

01004	n/a	Select or fetch into host variable that is too short.
02000 n/a	n/a	No data found. Equivalent to SQLCODE = 100. Occurs when:
		• Cursor fetch at end of result set.
		• Cursor fetch on empty result set.
		<ul> <li>Singleton select returns no data.</li> </ul>
		<ul> <li>Searched delete affects no rows.*</li> </ul>
		<ul> <li>Insert/select affects no rows.*</li> </ul>
		<ul> <li>Searched update affects no rows.*</li> </ul>
		* These conditions are only detected when the program is compiled with the NOSQLACCESS option to NSQLPREP.
07002	n/a	Number of columns does not match number of host variables.
21000	512	Singleton select returns more than one row, or subquery preceded by = returns more than one value.
22001	8152	String data truncated (on right) on insert or update.

22002	n/a	Null value returned with no indicator variable provided.
22003	168,220,232,234, 236-238,244,246,248, 519-524,535,8115, 10015 (DB-Library)	Arithmetic overflow error.
22005	206,235,245,247,249, 256,257,305,409,518, 529	Data type conversion error.
22008	210,211,241,242,295, 296,517	Data type conversion error.
22012	8134	Divide by zero error.
22019	1010, 506	Invalid escape character.
22022	n/a	Indicator variable overflow.
22024	n/a	Unterminated C string passed as input host variable.
22025	310	Invalid escape sequence, for example, escape char not followed by %, _, or escape char.
23000	233,272,273,515,530, 547,1505,1508,2601, 2615,2626,2627,3604, 3605	Constraint violation.
24000	n/a	Invalid cursor state.
25000	266,277,611,628,3902, 3903,3906,3908,6401	Transaction state error.
40001	1205,1211,2625, 3309,7112	Transaction rollback due to deadlock.
42000	207,208,213,229, 230,260	Syntax error, permission violation, or other nonspecific error.
44000	550	View with check option violation.

# **Embedded SQL for C Samples**

By default, Microsoft® SQL Server<sup>TM</sup> 2000 Setup installs the Embedded SQL for C (ESQL/C) sample source code in *x*:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\Esqlc.

Before you build any of the sample programs, use the Setenv.bat batch file to add the SQL Server development file header and library directories to the INCLUDE and LIB environment variables. Edit the Setenv.bat file to set the variables as appropriate.

You can build any of the sample programs using the supplied, general purpose makefile. The single makefile, Makefile, contains everything required to build an Embedded SQL program.

The default values for Makefile arguments support building the samples as Microsoft Win32® console applications. Makefile arguments can be supplied in the command prompt, or they can be set as environment variables for rapid processing of several example source files. For example, SET SERVER=*MyServer* could be used to cause **nsqlprep** to use *MyServer* as the build target for ESQL/C stored procedure creation.

Makefile	Default	
argument	value	Description
APP		Name of the sample source file to build. Do not include a file extension.
UTIL		Name of the utility source file. Do not include a file extension. The default value for the utility file is based on the value of the SUBSYS argument.
ENV	dos	Build sample for execution in Microsoft MS- DOS®.
	win16	Build sample as a 16-bit Microsoft Windows® application.
	qwin	Build console applications as 16-bit Windows, QuickWin applications.

	win32i	Default. Build applications for 32-bit Windows
		operating systems.
MODEL	medium	When building a sample for MS-DOS or 16-bit
		Windows, build the sample for the medium
		memory model.
	large	Default. When building the sample for MS-
		DOS or 16-bit Windows, build the sample for
		large memory model.
SUBSYS	console	Default. Build sample as a console (command
		window) application. Specifying console
		causes the UTIL argument to default to gcutil.
	windows	Build the sample as a Windows application.
		Specifying windows causes the UTIL argument
		to default to gwutil.
DEBUG	0	Default. Build a release version of the sample.
		The executable does not contain debugger
		support.
	1	Build a debug version of the sample.
SQLACCESS	0	Do not attempt to connect to a server running
		SQL Server and do not create stored
		procedures to support SQL statements
		embedded in C source code.
		Attempt to connect to an instance of SQL
		Server and create stored procedures.
	1	The default value of SQLACCESS is based on
		the presence or absence of a LOGIN argument
		value. Specify SQLACCESS to override the
		default assignment.
BIND	0	Default. Do not create a bind file.
	1	Create a bind file.
SERVER		Name of an instance of SQL Server to use for
		stored procedure creation.
LOGIN		SQL Server login account. The identifier is
		used by <b>nsqlprep</b> to connect to the server for
		stored procedure creation.

PASSWORD	SQL Server password. Used by <b>nsqlprep</b> to connect to the server.
DATABASE	SQL Server database <b>nsqlprep</b> will use when creating stored procedures. The default is <b>pubs</b> .

For information about nmake command prompts that build a particular example, see the documentation for that example.

When started, each sample application accepts the following arguments for specifying an instance of SQL Server: the login ID, password, and database to use. Each sample application generates a prompt for unspecified options.

#### Syntax

```
program [{/s | /S} [server_name]]
[{/u | /U} login_id]
[{/p | /P} [password]]
[{/d | /D} database]
```

#### Arguments

program

Is the sample program started.

server\_name

Is the name of an instance of SQL Server to connect to. If *server\_name* is omitted, the local SQL Server instance is used.

```
login_id
```

Is the login ID of an instance of SQL Server to connect to.

password

Is the password of an instance of SQL Server to connect to. If *password* is omitted, a NULL password is assumed.

database

Is the database to use.

# **GENCHAR (Generic for C)**

Genchar is a generic Embedded SQL for C (ESQL/C) character-mode program written using plain C, standard C libraries, and Embedded SQL statements. Genchar connects to a specified instance of Microsoft® SQL Server<sup>™</sup> 2000 and issues a SELECT statement query from the **authors** table in the **pubs** database.

#### **Default Location**

x:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\Esqlc

#### **Running the Sample**

You can build Genchar for the following environments using the associated commands.

Environment	Command
Microsoft Windows® 2000,	nmake -f "makefile." APP="genchar"
Microsoft Windows NT® 4.0	
and	
Microsoft Windows 95/98	
Microsoft MS-DOS®	nmake -f "makefile." APP="genchar"
	ENV="dos"
QuickWin	nmake -f "makefile." APP="genchar"
	ENV="qwin"

#### Remarks

Genchar uses these files.

Gcutil.c	Genchar.h
Gcutil.h	Quickwin.def
Genchar.sqc	

See Also

<u>Samples</u>

# Genwin (Generic for Windows NT, Windows 95, Windows 98, and 16-bit Windows)

Genwin is a generic Embedded SQL for C (ESQL/C) program for Microsoft® Windows® 2000, Microsoft Windows NT® 4.0, Microsoft Windows 95, and Microsoft Windows 98, and 16-bit Windows written using the Win16/Microsoft Win32® API, C language, standard libraries for 16-bit Windows and C, and Embedded SQL statements. Genchar connects to a specified instance of Microsoft SQL Server<sup>™</sup> 2000 and issues a SELECT statement query from the **authors** table in the **pubs** database.

#### **Default Location**

x:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\Esqlc

#### **Running the Sample**

You can build Genwin for the following environments using the associated commands.

Environment	Command
Microsoft Windows 2000,	nmake -f "makefile." APP="genwin"
Microsoft Windows NT 4.0,	SUBSYS="windows"
Microsoft Windows 95, and	
Microsoft Windows 98	
16-bit Windows	nmake -f "makefile." APP="genwin"
	SUBSYS="windows" ENV="win16"

#### Remarks

Genwin uses these files.

Genwin.def	Genwin.sqc
Genwin.h	Gwutil.c

Genwin.ico	Gwutil.h
Genwin.rc	

## See Also

Samples

# Edblib (Embedded SQL and DB-Library for Windows NT, Windows 95, Windows 98, and 16-bit Windows)

Edblib is an Embedded SQL for C (ESQL/C) and DB-Library program for Microsoft® Windows® 2000, Microsoft Windows NT® 4.0, Microsoft Windows 95, and Microsoft Windows 98, and 16-bit Windows written using the Win16/Microsoft Win32® API, C language, standard libraries for 16-bit Windows and C, Embedded SQL statements, and DB-Library functions. Edblib connects to a specified instance of Microsoft SQL Server<sup>™</sup> 2000 and issues a SELECT statement query from the **authors** table in the **pubs** database using ESQL/C and DB-Library. It uses the GET CONNECTION statement to obtain the DB-Library DBPROCESS connection pointer.

## **Default Location**

x:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\Esqlc

## **Running the Sample**

You can build Edblib for the following environments by using the associated commands.

Environment	Command
Microsoft Windows 2000,	nmake -f "makefile." APP="edblib"
Microsoft Windows NT 4.0	SUBSYS="windows"
and	
Microsoft Windows 95/98	
16-bit Windows	nmake -f "makefile." APP="edblib"
	SUBSYS="windows" ENV="win16"

#### Remarks

Edblib uses these files.

Edblib.def	Edblib.sqc
Edblib.h	Gwutil.c
Edblib.ico	Gwutil.h
Edblib.rc	

## See Also

<u>Samples</u>

# **Embedded SQL for C Examples**

The examples, from 1 through 8, describe Embedded SQL for C (ESQL/C) character-mode programs written using plain C, standard C libraries, and Embedded SQL statements. Each program connects to a specified instance of Microsoft® SQL Server<sup>™</sup> 2000 and executes a series of Embedded SQL statements.

#### **Default Location**

x:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\Esqlc

#### **Running the Sample**

You can build an example program for the following environments using the associated commands. *N* is the number of the example program.

Environment	Command
Microsoft Windows® 2000,	nmake -f "makefile." APP="example <i>n</i> "
Microsoft Windows NT® 4.0,	
Microsoft Windows 95,	
Microsoft Windows 98	
MS-DOS	nmake -f "makefile." APP="example <i>n</i> "
	ENV="dos"
QuickWin	nmake -f "makefile." APP="example <i>n</i> "
	ENV="qwin"

## Remarks

Each program described in the following examples uses the appropriate Example*n*.sqc, Gcutil.c, Gcutil.h, and Quickwin.def files. Some of these programs use the **author2** table, which you can create by using the Author2.sql file.

#### Example 1

- Declares and opens a standard cursor using a prepared SQL statement.
- Fetches rows from the **sysobjects** table in the specified database.

#### Example 2

- Declares and opens a browse cursor.
- Fetches rows from the **sales** table in the **pubs** database into a C structure.

## Example 3

- Illustrates error handling.
- Attempts to insert a row into the **authors** table that violates the constraint on the **au\_id** column.
- Reports the CHECK constraint violation.

## Example 4

- Connects to two instances of Microsoft® SQL Server<sup>™</sup> 2000, and declares and opens two standard cursors using a prepared SQL statement.
- Fetches rows from the **author2** table in the **pubs** database of each SQL Server.
- Allows you to issue positioned updates of each author's last name as the rows are fetched.

#### Example 5

- Issues a single SELECT statement to retrieve a row from the **titles** table in the **pubs** database into a C structure, or into the individual fields of a C structure.
- Uses browse cursors with a SELECT statement and the **sp\_who** stored procedure.
- Issues static and dynamic INSERT statements.
- Issues a SELECT statement of various date and time values.

#### **Example 6**

- Issues multiple static SQL statements within a transaction in a Transact-SQL batch.
- Executes the **sp\_addtype** and **sp\_droptype** stored procedures.
- Uses a browse cursor with the **sp\_who** stored procedure.
- Issues a SELECT statement into a temporary table and retrieves a row from that temporary table by using the EXECUTE IMMEDIATE statement.
- Executes a static Transact-SQL batch that uses control-of-flow language.

#### **Example 7**

• Opens a standard cursor by using HOLDLOCK within a transaction; inserts the cursor.

- Updates and deletes rows from the **authors** table in the **pubs** database.
- Inserts rows into the **authors** table within a transaction and then rolls back that transaction.
- Demonstrates that triggers are enforced by attempting to delete a row from the **titles** table that fires the **deltitle** trigger.
- Uses a browse cursor with the **sp\_who** stored procedure.
- Executes the **sp\_addtype** and **sp\_droptype** stored procedures.

#### **Example 8**

• Demonstrates behavior similar to that of **isql**.

#### See Also

**Samples**