CHAPTER 2

The Embedded SQL Interface

About this chapter

This chapter describes the Embedded SQL programming interface to Adaptive Server Anywhere.

Contents

Topic	Page
Application development using Embedded SQL	8
Embedded SQL data types	17
Using host variables	20
The SQL Communication Area	27
Fetching data	32
Static and dynamic SQL	37
The SQL descriptor area (SQLDA)	45
Using stored procedures in Embedded SQL	51
Library functions	55
Embedded SQL commands	70
Database examples	72

Application development using Embedded SQL

Embedded SQL consists of SQL statements intermixed with C or C++ source code. These SQL statements are translated by a **SQL preprocessor** into C or C++ source code. The SQL preprocessor is run before compilation.

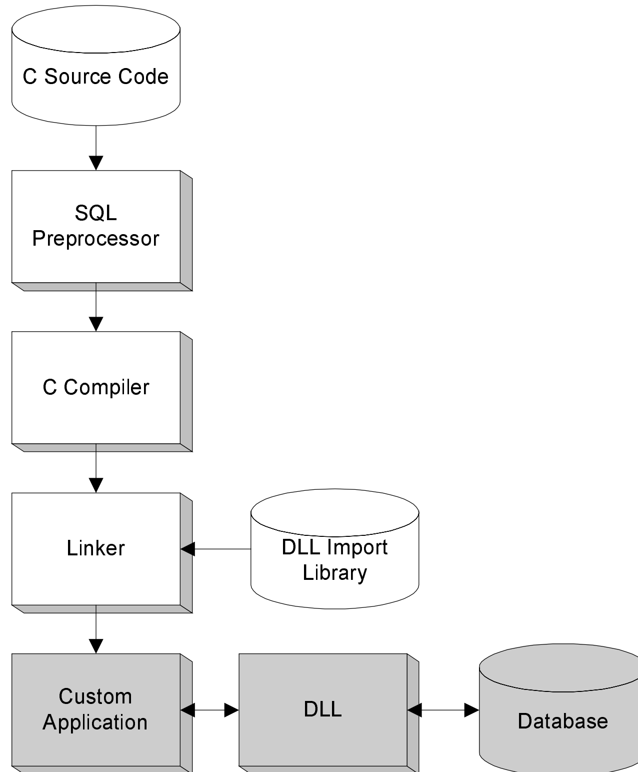
This code, together with the Adaptive Server Anywhere **interface library** communicates the appropriate information to the database server when the program is run. The interface library is a dynamic link library (**DLL**) or shared library on most platforms.

Supported compilers

The C language SQL preprocessor has been used in conjunction with the following compilers:

Operating system	Compiler	Version
Windows 95 and NT	Watcom C/C++	9.5 and above
Windows 95 and NT	Microsoft Visual C/C++	1.0 and above
Windows 95 and NT	Borland C++	4.5
Windows 3.x	Watcom C/C++	9.0 and above
Windows 3.x	Microsoft C / C++	5.0, 5.1, 6.0, 7.0
Windows 3.x	Microsoft Visual C/C++	1.0, 1.5
Windows 3.x	Borland C++	2.0, 3.0, 4.0, 4.5
UNIX	GNU gcc, native compiler	

Development process overview



Once the program has been successfully preprocessed and compiled, it is linked with the **import library** for the Adaptive Server Anywhere interface library to form an executable file. When the database is running, this executable file will use the Adaptive Server Anywhere DLL to interact with the database. The database does not have to be running when the program is preprocessed.

One Windows 3.x import library works for all compilers and memory models. For Windows 95 and Windows NT, there are separate import libraries for Watcom C/C++, for Microsoft Visual C++, and for Borland C++.

Watcom C/C++ supports 32-bit application development under Windows 3.x. For this environment, a static interface library (not a DLL) is provided.

Using import libraries is the standard development method for applications that call functions in DLLs. Adaptive Server Anywhere also provides an alternative, and recommended, method, which avoids the use of import libraries. For more information, see "Loading the interface library dynamically" on page 13.

Running the SQL preprocessor

The SQL preprocessor is an executable named *sqlpp.exe*.

Command line

The SQLPP command line is as follows:

SQLPP [*switches*] *sql-filename* [*output-filename*]

The SQL preprocessor processes a C program with Embedded SQL before the C or C++ compiler is run. The preprocessor translates the SQL statements into C/C++ language source that is put into the output file. The normal extension for source programs with Embedded SQL is *.sqc*. The default output filename is the *sql-filename* with an extension of *.c*. If the *sql-filename* already has a *.c* extension, then the output filename extension is *.cc* by default.

For a full listing of the command-line switches, see "The SQL preprocessor" on page 124 of the book *Adaptive Server Anywhere Reference Manual*.

Embedded SQL header files

All header files are installed in the *.h* subdirectory of your Adaptive Server Anywhere installation directory.

Filename	Description
sqlca.h	Main header file included in all Embedded SQL programs. This file includes the structure definition for the SQL Communication Area (SQLCA) and prototypes for all Embedded SQL database interface functions.
sqlda.h	SQL Descriptor Area structure definition included in Embedded SQL programs that use dynamic SQL.
sqldef.h	Definition of Embedded SQL interface data types. This file also contains structure definitions and return codes needed for starting the database server from a C program.
sqlerr.h	Definitions for error codes returned in the **sqlcode** field of the SQLCA.
sqlstate.h	Definitions for ANSI/ISO SQL standard error states returned in the **sqlstate** field of the SQLCA.
pshpk1.h, pshpk2.h, poppk.h	These headers ensure that structure packing is handled correctly. They support Watcom C/C++, Microsoft Visual C++, IBM Visual Age, and Borland C/C++ compilers.

Import libraries

All import libraries are installed in the *lib* subdirectory, under the operating system subdirectory of the Adaptive Server Anywhere installation directory. For example, Windows 95 and Windows NT import libraries are stored in the *win32\lib* subdirectory.

Operating system	Compiler	Import library
Windows 95 and NT	Watcom C/C++	*dblibwfw.lib*
Windows 95 and NT	Watcom C/C++ stack calling convention.	*dblibfws.lib*
Windows 95 and NT	Watcom C/C++	*dblibtw.lib*
Windows 95 and NT	Borland C++	*dblibtb.lib*
Windows 95 and NT	Microsoft Visual C++	*dblibtm.lib*
Windows 3.x	All compilers	*dblibw.lib*

A simple example

The following is a very simple example of an Embedded SQL program.

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
   db_init( &sqlca );
   EXEC SQL WHENEVER SQLERROR GOTO error;
   EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
   EXEC SQL UPDATE employee
      SET emp_lname =    'Plankton'
      WHERE emp_id = 195;
   EXEC SQL COMMIT WORK;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
   return( 0 );

   error:
   printf( "update unsuccessful -- sqlcode = %ld.n",
   sqlca.sqlcode );
   return( -1 );
}
```

This example connects to the database, updates the surname of employee number 1056, commits the change and exits. There is virtually no interaction between the SQL and C code. The only thing the C code is used for in this example is control flow. The WHENEVER statement is used for error checking. The error action (GOTO in this example) is executed after any SQL statement that causes an error.

Note that the first section of this chapter uses UPDATE and INSERT examples because they are simpler.

For a description of fetching data, see "Fetching data" on page 32.

Structure of Embedded SQL programs

SQL statements are placed (embedded) within regular C or C++ code. All Embedded SQL statements start with the words EXEC SQL and end with a semicolon (;). Normal C language comments are allowed in the middle of Embedded SQL statements.

Every C program using Embedded SQL must contain the following statement before any other Embedded SQL statements in the source file.

```
EXEC SQL INCLUDE SQLCA;
```

The first Embedded SQL statement executed by the C program must be a CONNECT statement. The CONNECT statement is used to establish a connection with the database server and to specify the user ID that is used for authorizing all statements executed during the connection.

The CONNECT statement must be the first Embedded SQL statement executed. Some Embedded SQL commands do not generate any C code, or do not involve communication with the database. These commands are thus allowed before the CONNECT statement. Most notable are the INCLUDE statement and the WHENEVER statement for specifying error processing.

User IDs and database authorization

The CONNECT statement specifies a user ID and password to be used for checking the permissions for any dynamic SQL statements that are used in the program (defined in "Dynamic SQL statements" on page 37). It is also used for authorization of any static statements that are contained in modules preprocessed without the SQL preprocessor `-l` option. The `-l` option provides user identification.

Static SQL statements in modules that are preprocessed with the `-l` option will have the authorization checked at execution time using the user ID and password specified with the `-l` option. This is how privileged commands can be executed by non-privileged users under C program control.

Loading the interface library dynamically

The usual practice for developing applications that use functions from DLLs is to link the application against an **import library**, which contains the required function definitions.

This section describes an alternative to using an import library for developing Adaptive Server Anywhere applications. The Adaptive Server Anywhere interface library can be loaded dynamically, without having to link against the import library, using the *esqldll.c* module in the *src* subdirectory of your installation directory. Using *esqldll.c* is recommended, because it is easier to use and more robust in its ability to locate the interface DLL.

❖ **To load the interface DLL dynamically:**

1. Your program must call **db_init_dll** to load the DLL, and must call **db_fini_dll** to free the DLL. The **db_init_dll** call must be before any function in the database interface, and no function in the interface can be called after **db_fini_dll**.

You must still call the **db_init** and **db_fini** library functions.

2 You must **#include** the *esqldll.h* header file before the EXEC SQL INCLUDE SQLCA statement or **#include** <*sqlca.h*> line in your Embedded SQL program.

3 A SQL OS macro must be defined. The header file *sqlca.h*, which is included by *esqdll.c*, attempts to determine the appropriate macro and defines it. However, certain combinations of platforms and compilers may cause this to fail. In this case, you must add a **#define** to the top of this file, or make the definition by using a compiler option.

Macro	Platforms
_SQL_OS_WINNT	Windows 95 and Windows NT
_SQL_OS_WINDOWS	Windows 3.x

4 Compile *esqldll.c*.

5 Instead of linking against the imports library, link the object module *esqldll.obj* with your Embedded SQL application objects.

Example

The following example illustrates how to use *esqldll.c* and *esqldll.h*.

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

EXEC SQL INCLUDE SQLCA;
#include "sqldef.h"
#include <windows.h>

EXEC SQL BEGIN DECLARE SECTION;
int          x;
a_sql_statement_number       stat1;
EXEC SQL END DECLARE SECTION;

#define TRUE 1
#define FALSE 0


void printSQLError( void )
{
   char      buffer[200];

   sqlerror_message( &sqlca, buffer, sizeof(buffer) );
   #ifdef _SQL_OS_WINDOWS
   printf( "Error %ld -- %Fs\n", SQLCODE, buffer );
   #else
   printf( "Error %ld -- %s\n", SQLCODE, buffer );
   #endif
}
```

```
char *dllpaths[] = { "s:\\jasonhi\\",
          NULL };


#include <windows.h>

int main( void )
{

   struct sqlda _fd_ *   sqlda1;
   int      result;
   char     string[200];

   printf( "Initing DLL\n" );

   result = db_init_dll( dllpaths );
   switch( result ) {
   case ESQLDLL_OK:
      printf("OK\n");
      break;
      case ESQLDLL_DLL_NOT_FOUND:
      printf("DLL NOT FOUND\n");
      return( 10 );
   case ESQLDLL_WRONG_VERSION:
      printf("WRONG VERSION\n");
      return( 10 );
   }
   if( !db_init( &sqlca ) ) {
      printf("db_init failed.\n");
      db_fini_dll();
   return( 10 );
   }

   #ifdef _SQL_OS_WINNT
   result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=d:\\asa6\\sample.db");

   #elif defined( _SQL_OS_WINDOWS )
   result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=c:\\wsql50\\sample.db");

   #elif defined( _SQL_OS_OS232 )
   result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=h:\\wsql50\\sample.db");

   #endif

   if( ! result ) {
      printf( "db_string_connect returned = %d\n",
result );
   printSQLError();
```

```
    }
    sqlda1 = alloc_descriptor( sqlcaptr, 20 );
    EXEC SQL PREPARE :stat1 FROM
       'select * from employee
       WHERE empnum = 80921';
    if( SQLCODE != 0 ) {
       printSQLError();
    }
    EXEC SQL DECLARE curs CURSOR FOR :stat1;
    if( SQLCODE != 0 ) {
       printSQLError();
    }
    EXEC SQL OPEN curs;
    if( SQLCODE != 0 ) {
       printSQLError();
    }
    EXEC SQL DESCRIBE :stat1 INTO sqlda1;
    if( SQLCODE != 0 ) {
       printSQLError();
    }
    sqlda1->sqlvar[1].sqltype = 460;
    fill_sqlda( sqlda1 );
    EXEC SQL FETCH FIRST curs INTO DESCRIPTOR sqlda1;
    if( SQLCODE != 0 ) {
       printSQLError();
    }
    printf( "name = %Fs\n", (char _fd_ *)sqlda1-
>sqlvar[1].sqldata );
    x = sqlda1->sqld;
    printf( "COUNT = %d\n", x );

    free_filled_sqlda( sqlda1 );
    db_string_disconnect( &sqlca, "" );
    db_fini( &sqlca );
    db_fini_dll();

    return( 0 );
}
```

Embedded SQL data types

To transfer information between a program and the database server, every piece of data must have a data type. The Embedded SQL data type constants are prefixed with DT_, and can be found in the *sqldef.h* header file. You can create a host variable of any one of the supported types. You can also use these types in a SQLDA structure for passing data to and from the database.

The following data types are supported by the Embedded SQL programming interface:

- **DT_SMALLINT** 16 bit, signed integer.
- **DT_INT** 32 bit, signed integer.
- **DT_FLOAT** 4 byte floating point number.
- **DT_DOUBLE** 8 byte floating point number.
- **DT_DECIMAL** Packed decimal number.

```
typedef struct DECIMAL {
char array[1];
 } DECIMAL;
```

- **DT_STRING** NULL-terminated blank-padded character string.
- **DT_DATE** NULL-terminated character string that is a valid date.
- **DT_TIME** NULL-terminated character string that is a valid time.
- **DT_TIMESTAMP** NULL-terminated character string that is a valid timestamp.
- **DT_FIXCHAR** Fixed-length blank padded character string.
- **DT_VARCHAR** Varying length character string with a two byte length field. When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field (not padded).

```
typedef struct VARCHAR {
unsigned short int len;
char array[1];
} VARCHAR;
```

- **DT_BINARY** Varying length binary data with a two byte length field. When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field.

```
typedef struct BINARY {
unsigned short int len;
char array[1];
```

```
} BINARY;
```

- **DT_TIMESTAMPSTRUCT** SQLDATETIME structure with fields for each part of a timestamp.

  ```
  typedef struct sqldatetime {
  unsigned short year; /* e.g. 1992 */
  unsigned char month; /* 0-11 */
  unsigned char day_of_week; /* 0-6 0=Sunday */
  unsigned short day_of_year; /* 0-365 */
  unsigned char day; /* 1-31 */
  unsigned char hour; /* 0-23 */
  unsigned char minute; /* 0-59 */
  unsigned char second; /* 0-59 */
  unsigned long microsecond; /* 0-999999 */
  } SQLDATETIME;
  ```

  The SQLDATETIME structure can be used to retrieve fields of DATE, TIME, and TIMESTAMP type (or anything that can be converted to one of these). Often, applications have their own formats and date manipulation code. Fetching data in this structure makes it easier for a programmer to manipulate this data. Note that DATE, TIME and TIMESTAMP fields can also be fetched and updated with any character type.

  For more information, see the DATE_FORMAT, TIME_FORMAT, TIMESTAMP_FORMAT, and DATE_ORDER database options in "Database Options" on page 127 of the book *Adaptive Server Anywhere Reference Manual*.

- **DT_VARIABLE** NULL-terminated character string. The character string must be the name of a SQL variable whose value is used by the database server. This data type is used only for supplying data to the database server. It cannot be used when fetching data from the database server.

The structures are defined in the *sqlca.h* file. The VARCHAR, BINARY and DECIMAL types contain a one-character array and are thus not useful for declaring host variables but they are useful for allocating variables dynamically or typecasting other variables.

DATE and TIME database types

There are no corresponding Embedded SQL interface data types for the various DATE and TIME database types. These database types are all fetched and updated using either the SQLDATETIME structure or character strings.

There are no Embedded SQL interface data types for LONG VARCHAR and LONG BINARY database types. These database types are fetched and updated in pieces.

☞ For more information see "GET DATA statement" on page 479 of the book *Adaptive Server Anywhere Reference Manual* and "SET statement" on page 546 of the book *Adaptive Server Anywhere Reference Manual*.

Using host variables

Host variables are C variables that are identified to the SQL preprocessor. Host variables can be used to send values to the database server or receive values from the database server.

Host variables are quite easy to use, but they have some restrictions. Dynamic SQL is a more general way of passing information to and from the database server using a structure known as the SQL Descriptor Area (SQLDA). Dynamic SQL is discussed in "Static and dynamic SQL" on page 37.

Declaring host variables

Host variables are defined by putting them into a **declaration section**. According to the IBM SAA and ANSI Embedded SQL standards, host variables are defined by surrounding the normal C variable declarations with the following:

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

These host variables can then be used in place of value constants in any SQL statement. When the database server executes the command, the value of the host variable is used. Note that host variables cannot be used in place of table or column names; dynamic SQL is required for this. The variable name is prefixed with a colon (:) in a SQL statement to distinguish it from other identifiers allowed in the statement.

A standard SQL preprocessor does not scan C language code except inside a DECLARE SECTION. Thus, TYPEDEF types and structures are not allowed. Initializers on the variables are allowed inside a DECLARE SECTION.

Example

- The following sample code illustrates the use of host variables on an INSERT command. The variables are filled in by the program and then inserted into the database:

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate
values
*/
```

```
EXEC SQL INSERT INTO Employee
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

For a more extensive example, see "Static cursor example" on page 74.

C host variable types

Only a limited number of C data types are supported as host variables. Also, certain host variable types do not have a corresponding C type.

There are macros defined in the *sqlca.h* header file that can be used to declare a host variable of these types: VARCHAR, FIXCHAR, BINARY, PACKED DECIMAL, or SQLDATETIME structure. They are used as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_VARCHAR( 10 ) v_varchar;
DECL_FIXCHAR( 10 ) v_fixchar;
DECL_BINARY( 4000 ) v_binary;
DECL_DECIMAL( 10, 2 ) v_packed_decimal;
DECL_DATETIME v_datetime;
EXEC SQL END DECLARE SECTION;
```

The preprocessor recognizes these macros within a declaration section and will treat the variable as the appropriate type.

The following table lists the C variable types that are allowed for host variables, and their corresponding Embedded SQL interface data types.

C Data Type	Embedded SQL Interface Type	Description

`short i;`
`short int i;`
`unsigned short int i;`
	DT_SMALLINT	16 bit, signed integer

`long l;`
`long int l;`
`unsigned long int l;`
	DT_INT	32 bit, signed integer

`float f;`	DT_FLOAT	4 byte floating point
`double d;`	DT_DOUBLE	8 byte floating point
`DECL_DECIMAL(p,s)`	DT_DECIMAL(p,s)	Packed decimal
`char a; /*n=1*/` `DECL_FIXCHAR(n) a; DECL_FIXCHAR a[n];`	DT_FIXCHAR(n)	Fixed length character string blank padded

C Data Type	Embedded SQL Interface Type	Description
`char a[n]; /*n>=1*/`	DT_STRING(n)	NULL-terminated blank-padded string
`char *a;`	DT_STRING(32767)	NULL-terminated string
`DECL_VARCHAR(n) a;`	DT_VARCHAR(n)	Varying length character string with 2 byte length field. Not blank padded.
`DECL_BINARY(n) a;`	DT_BINARY(n)	Varying length binary data with 2 byte length field
`DECL_DATETIME a;`	DT_TIMESTAMP_STRUCT	SQLDATETIME structure

Pointers to char

A host variable declared as a **pointer to char** (*char *a*) is considered by the database interface to be 32,767 bytes long. Any host variable of type **pointer to char** used to retrieve information from the database must point to a buffer large enough to hold any value that could possibly come back from the database.

This is potentially quite dangerous, because somebody could change the definition of the column in the database to be larger than it was when the program was written. This could cause random memory corruption problems. If you are using a 16-bit compiler, requiring 32,767 bytes could make the program stack overflow. It is better to use a declared array, even as a parameter to a function where it is passed as a **pointer to char**. This lets the PREPARE statements know the size of the array.

Scope of host variables

A standard host variable declaration section can appear anywhere that C variables can normally be declared. This includes the parameter declaration section of a C function. The C variables have their normal scope (available within the block in which they are defined). However, since the SQL preprocessor does not scan C code, it does not respect C blocks.

As far as the SQL preprocessor is concerned, host variables are global; two host variables cannot have the same name. The only exception to this rule is that two host variables can have the same name if they have identical types (including any necessary lengths).

Host variable usage

Host variables can be used in the following circumstances:

- SELECT, INSERT, UPDATE and DELETE statements in any place where a number or string constant is allowed.
- The INTO clause of SELECT and FETCH statements.
- Host variables can also be used in place of a statement name, a cursor name, or an option name in commands specific to Embedded SQL.
- For CONNECT, DISCONNECT, and SET CONNECT, a host variable can be used in place of a user ID, password, connection name, or database environment name.
- For SET OPTION and GET OPTION, a host variable can be used in place of a user ID, option name or option value.
- Host variables cannot be used in place of a table name or a column name in any statement.

Examples

- The following is a valid program:

```
INCLUDE SQLCA;
long SQLCODE;
sub1() {
   char SQLSTATE[6];
   exec sql CREATE TABLE ...
}
```

- The following is an invalid program:

```
INCLUDE SQLCA;
sub1() {
   char SQLSTATE[6];
   exec sql CREATE TABLE...
}
sub2() {
   exec sql DROP TABLE...
   // No SQLSTATE in scope of this statement
}
```

- The case of SQLSTATE and SQLCODE is important, and the ISO/ANSI standard requires that their definitions be exactly as follows:

```
long SQLCODE;
char SQLSTATE[6];
```

Indicator variables

Indicator variables are C variables that hold supplementary information when you are fetching or putting data. There are several distinct uses for indicator variables:

- **NULL values** To enable applications to handle NULL values.

- **String truncation** To enable applications to handle cases when fetched values must be truncated to fit into host variables.
- **Conversion errors** To hold error information.

An indicator variable is a host variable of type **short int** that is placed immediately following a regular host variable in a SQL statement. For example, in the following INSERT statement, **:ind_phone** is an indicator variable:

```
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
   :employee_initials, :employee_phone:ind_phone );
```

Using indicator variables to handle NULL

In SQL data, NULL represents either an unknown attribute or inapplicable information. The SQL concept of NULL is not to be confused with the C language constant by the same name (**NULL**). The C constant is used to represent a non-initialized or invalid pointer.

When NULL is used in the Adaptive Server Anywhere documentation, it refers to the SQL database meaning given above. The C language constant is referred to as the **null** pointer (lower case).

NULL is not the same as any value of the column's defined type. Thus, in order to pass NULL values to the database or receive NULL results back, something extra is required beyond regular host variables. **Indicator variables** are used for this purpose.

Using indicator variables when inserting NULL

An INSERT statement could include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;

/*
program fills in empnum, empname,
initials and homephone
*/
if( /* phone number is unknown */ ) {
   ind_phone = -1;
} else {
   ind_phone = 0;
}
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
```

```
        :employee_initials, :employee_phone:ind_phone );
```

If the indicator variable has a value of -1, a NULL is written. If it has a value of 0, the actual value of **employee_phone** is written.

Using indicator variables when fetching NULL

Indicator variables are also used when receiving data from the database. They are used to indicate that a NULL value was fetched (indicator is negative). If a NULL value is fetched from the database and an indicator variable is not supplied, an error is generated (SQLE_NO_INDICATOR). Errors are explained in the next section.

Using indicator variables for truncated values

Indicator variables indicate whether any fetched values were truncated to fit into a host variable. This enables applications to handle truncation appropriately.

If a value is truncated on fetching, the indicator variable is set to a positive value, containing the actual length of the database value before truncation. If the length of the value is greater than 32,767, then the indicator variable contains 32,767.

Using indicator values for conversion errors

By default, the CONVERSION_ERROR database option is set to ON, and any data type conversion failure leads to an error, with no row returned.

You can use indicator variables to tell which column produced a data type conversion failure. If you set the database option CONVERSION_ERROR to OFF, any data type conversion failure gives a warning, rather than an error. If the column that suffered the conversion error has an indicator variable, that variable is set to a value of -2.

If you set the CONVERSION_ERROR option to OFF when inserting data into the database, a value of NULL is inserted when a conversion failure occurs.

Summary of indicator variable values

The following table provides a summary of indicator variable usage.

Indicator Value	Supplying Value to database	Receiving value from database
> 0	Host variable value	Retrieved value was truncated — actual length in indicator variable
0	Host variable value	Fetch successful, or CONVERSION_ERROR set to ON.
-1	NULL value	NULL result
-2	NULL value	Conversion error (when CONVERSION_ERROR is set to OFF only). SQLCODE will indicate a conversion error.
< -2	NULL value	NULL result

For more information on retrieving long values, see "GET DATA statement" on page 479 of the book *Adaptive Server Anywhere Reference Manual*.

The SQL Communication Area

The **SQL Communication Area** (**SQLCA**) is an area of memory that is used on every database request for communicating statistics and errors from the application to the database server and back to the application. The SQLCA is used as a handle for the application-to-database communication link. It is passed in to all database library functions that need to communicate with the database server. It is implicitly passed on all Embedded SQL statements.

A global SQLCA variable is defined in the interface library. The preprocessor generates an external reference for the global SQLCA variable and an external reference for a pointer to it. The external reference is named **sqlca** and is of type SQLCA. The pointer is named **sqlcaptr**. The actual global variable is declared in the imports library.

The SQLCA is defined by the *sqlca.h* header file, included in the *h* subdirectory of your installation directory.

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The **sqlcode** and **sqlstate** fields contain error codes when a database request has an error (see below). Some C macros are defined for referencing the **sqlcode** field, the **sqlstate** field, and some other fields.

Fields in the SQLCA

The fields in the SQLCA have the following meanings:

- **sqlcaid** An 8-byte character field that contains the string **SQLCA** as an identification of the SQLCA structure. This field helps in debugging when you are looking at memory contents.
- **sqlcabc** A long integer that contains the length of the SQLCA structure (136 bytes).
- **sqlcode** A long integer that specifies the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file *sqlerr.h*. The error code is 0 (zero) for a successful operation, positive for a warning and negative for an error.

  For a full listing of error codes, see "Database Error Messages" on page 581 of the book *Adaptive Server Anywhere Reference Manual*.
- **sqlerrml** The length of the information in the **sqlerrmc** field.
- **sqlerrmc** May contain one or more character strings to be inserted into an error message. Some error messages contain a placeholder string (*%1*) which is replaced with the text in this field.

For example, if a Table Not Found error is generated, **sqlerrmc** contains the table name, which is inserted into the error message at the appropriate place.

For a full listing of error messages, see "Database Error Messages" on page 581 of the book *Adaptive Server Anywhere Reference Manual*.

- **sqlerrp** Reserved.
- **sqlerrd** A utility array of long integers.
- **sqlwarn** Reserved.
- **sqlstate** The SQLSTATE status value. The ANSI SQL standard (SQL-92) defines a new type of return value from a SQL statement in addition to the SQLCODE value in previous standards. The SQLSTATE value is always a five-character null-terminated string, divided into a two character class (the first two characters) and a three-character subclass. Each character can be a digit from 0 through 9 or an upper case alphabetic character A through Z.

  Any class or subclass that begins with 0 through 4 or A through H is defined by the SQL standard; other classes and subclasses are implementation defined. The SQLSTATE value '00000' means that there has been no error or warning.

  Other SQLSTATE values are described in "Database Error Messages" on page 581 of the book *Adaptive Server Anywhere Reference Manual*.

sqlerror array

The **sqlerror** field array has the following elements.

- **sqlerrd[1] (SQLIOCOUNT)** The actual number of input/output operations that were required to complete a command.

  The database does not start this number at zero for each command. Your program can set this variable to zero before executing a sequence of commands. After the last command, this number is the total number of input/output operations for the entire command sequence.

- **sqlerrd[2] (SQLCOUNT)** The value of this field depends on which statement is being executed.
  - **INSERT, UPDATE and DELETE statements** The number of rows that were affected by the statement.

On a cursor OPEN, this field is filled in with either the actual number of rows in the cursor (a value greater than *or equal to* 0) or an estimate thereof (a negative number whose absolute value is the estimate). It is the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows using the ROW_COUNT option.

- **FETCH cursor statement** The SQLCOUNT field is filled if a SQLE_NOTFOUND warning is returned. It contains the number of rows by which a FETCH RELATIVE or FETCH ABSOLUTE statement goes outside the range of possible cursor positions. (A cursor can be on a row, before the first row or after the last row.)

  The value is 0 if the row was not found but the position is valid, for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value is positive if the attempted fetch was beyond the end of the cursor, and negative if the attempted fetch was before the beginning of the cursor.

- **GET DATA statement** The SQLCOUNT field holds the actual length of the value.

- **DESCRIBE statement** In the WITH VARIABLE RESULT clause used to describe procedures that may have more than one result set, SQLCOUNT is set to one of the following values:

  - **0** The result set may change: the procedure call should be described again following each OPEN statement.

  - **1** The result set is fixed. No redescribing is required.

  In the case of a syntax error, SQLE_SYNTAX_ERROR, this field contains the approximate character position within the command string where the error was detected.

- **sqlerrd[3] (SQLIOESTIMATE)** The estimated number of input/output operations that are to complete the command. This field is given a value on an OPEN or EXPLAIN command.

SQLCA management for multi-threaded or reentrant code

You can use Embedded SQL statements in multi-threaded or reentrant code. However, if you use a single connection, you are restricted to one active request per connection. In a multi-threaded application, you should not use the same connection to the database on each thread unless you use a semaphore to control access.

There are no restrictions on using separate connections on each thread that wishes to use the database. The SQLCA is used by the runtime library to distinguish between the different thread contexts. Thus, each thread wishing to use the database must have it's own SQLCA. Any given database connection will only be accessible from one SQLCA.

Using multiple SQLCAs

❖ **To manage multiple SQLCAs in your application:**

1. You must use the command line switch on the SQL preprocessor that generates reentrant code (`-r`). The reentrant code is a little larger and a little slower because statically initialized global variables cannot be used. However, these effects are minimal.
2. Each SQLCA used in your program must be initialized with a call to **db_init** and cleaned up at the end with a call to **db_fini**.

> ***Caution***
> *Failure to call db_fini for each db_init on NetWare can cause the database server to fail, and the NetWare file server to fail.*

3. The Embedded SQL statement SET SQLCA ("SET SQLCA statement" on page 557 of the book *Adaptive Server Anywhere Reference Manual*) is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as: EXEC SQL SET SQLCA 'task_data->sqlca'; is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data. This statement does not generate any code and thus has no performance impact. It changes the state within the preprocessor so that any reference to the SQLCA will use the given string.

For information about creating SQLCAs, see "SET SQLCA statement" on page 557 of the book *Adaptive Server Anywhere Reference Manual*.

When to use multiple SQLCAs

You can use the multiple SQLCA support in any of the supported Embedded SQL environments, but it is only required in reentrant code.

The following list details the environments where multiple SQLCAs must be used:

- **Multi-threaded applications** If more than one thread uses the same SQLCA, a context switch can cause more than one thread to be using the SQLCA at the same time. Each thread must have its own SQLCA. This can also happen when you have a DLL that uses Embedded SQL and is called by more than one thread in your application.
- **Dynamic link libraries and shared libraries** A DLL has only one data segment. While the database server is processing a request from one application, it may yield to another application that makes a request to the database server. If your DLL uses the global SQLCA, both applications are using it at the same time. Each Windows application must have its own SQLCA.
- **A DLL with one data segment** A DLL can be created with only one data segment or one data segment for each application. If your DLL has only one data segment, you cannot use the global SQLCA for the same reason, that a DLL cannot use the global SQLCA. Each application must have its own SQLCA.

Connection management with multiple SQLCAs

You do not need to use multiple SQLCAs to connect to more than one database or have more than one connection to a single database.

Each SQLCA can have one unnamed connection. Each SQLCA has an active or current connection (see "SET CONNECTION statement" on page 551 of the book *Adaptive Server Anywhere Reference Manual*). All operations on a given database connection must use the same SQLCA that was used when the connection was established.

Record locking
Operations on different connections are subject to the normal record locking mechanisms and may cause each other to block and possibly to deadlock. For information on locking, see the chapter "Using Transactions and Locks" on page 367 of the book *Adaptive Server Anywhere User's Guide*.

Fetching data

Fetching data in Embedded SQL is done using the SELECT statement. There are two cases:

- ♦ The SELECT statement returns at most one row.
- ♦ The SELECT statement may return multiple rows.

Embedded SELECT

A single row query retrieves at most one row from the database. A single-row query SELECT statement has an INTO clause following the select list and before the FROM clause. The INTO clause contains a list of host variables to receive the value for each select list item. There must be the same number of host variables as there are select list items. The host variables may be accompanied by indicator variables to indicate NULL results.

When the SELECT statement is executed, the database server retrieves the results and places them in the host variables. If the query results contain more than one row, the database server returns an error.

If the query results in no rows being selected, a Row Not Found warning is returned. Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area" on page 27.

Example

For example, the following code fragment returns 1 if a row from the employee table is successfully fetched, 0 if the row doesn't exist and -1 if an error occurs.

```
EXEC SQL BEGIN DECLARE SECTION;
   long         emp_id;
   char         name[41];
   char         sex;
   char         birthdate[15];
   short int ind_birthdate;
EXEC SQL END DECLARE SECTION;
. . .
int find_employee( long employee )
{
   emp_id = employee;
   EXEC SQL  SELECT emp_fname ||
             ' ' || emp_lname, sex, birth_date
             INTO :name, :sex,
                    birthdate:ind_birthdate
             FROM "dba".employee
             WHERE emp_id = :emp_id;
   if( SQLCODE == SQLE_NOTFOUND ) {
```

```
      return( 0 ); /* employee not found */
   } else if( SQLCODE < 0 ) {
      return( -1 ); /* error */
   } else {
      return( 1 ); /* found */
   }
}
```

Cursors in Embedded SQL

A cursor is used to retrieve rows from a query that has multiple rows in its result set. A **cursor** is a handle or an identifier for the SQL query and a position within the result set.

For an introduction to cursors, see "Working with cursors" on page 211 of the book *Adaptive Server Anywhere User's Guide*.

❖ **To manage a cursor in Embedded SQL:**

1 Declare a cursor for a particular SELECT statement, using the DECLARE statement.

2 Open the cursor using the OPEN statement.

3 Retrieve results one row at a time from the cursor using the FETCH statement.

4 Fetch rows until the Row Not Found warning is returned.

  Errors and warnings are returned in the SQLCA structure, described in "The SQL Communication Area" on page 27.

5 Close the cursor, using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK). Cursors that are opened with a WITH HOLD clause are kept open for subsequent transactions until they are explicitly closed.

The following is a simple example of cursor usage:

```
void print_employees( void )
{
   int status;
   EXEC SQL BEGIN DECLARE SECTION;
   char name[50];
   char sex;
   char birthdate[15];
   short int ind_birthdate;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL DECLARE C1 CURSOR FOR
```

```
        SELECT emp_fname || ' ' || emp_lname,
                 sex, birth_date
        FROM "dba".employee;
    EXEC SQL OPEN C1;
    for( ;; ) {
        EXEC SQL FETCH C1 INTO :name, :sex,
:birthdate:ind_birthdate;
        if( SQLCODE == SQLE_NOTFOUND ) {
            break;
        } else if( SQLCODE < 0 ) {
            break;
        }
        if( ind_birthdate < 0 ) {
            strcpy( birthdate, "UNKNOWN" );
        }
        printf( "Name: %s Sex: %c Birthdate:
                 %s.n",name, sex, birthdate );
    }
EXEC SQL CLOSE C1;
}
```

For complete examples using cursors, see "Static cursor example" on page 74, and "Dynamic cursor example" on page 75.

Cursor positioning

A cursor is positioned in one of three places:

- On a row
- Before the first row
- After the last row

Absolute row from start		Absolute row from end
0	Before first row	–n – 1
1		–n
2		–n + 1
3		–n + 2
n – 2		–3
n – 1		–2
n		–1
n + 1	After last row	0

When a cursor is opened, it is positioned before the first row. The cursor position can be moved, using the FETCH command (see "FETCH statement" on page 468 of the book *Adaptive Server Anywhere Reference Manual*). It can be positioned to an absolute position either from the start or from the end of the query results. It can also be moved relative to the current cursor position.

There are special *positioned* versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a No Current Row of Cursor error is returned.

The PUT statement can be used to insert a row into a cursor.

Cursor positioning problems

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.

With Adaptive Server Anywhere, this occurs if a temporary table had to be created to open the cursor (see "Temporary tables used in query processing" on page 640 of the book *Adaptive Server Anywhere User's Guide* for a description).

The UPDATE statement may cause a row to move in the cursor. This happens if the cursor has an ORDER BY clause that uses an existing index (a temporary table is not created).

Static and dynamic SQL

There are two ways to embed SQL statements into a C program:

- Static statements
- Dynamic statements

Until now, we have been discussing static SQL. This section compares static and dynamic SQL.

Static SQL statements

All standard SQL data manipulation and data definition statements can be embedded in a C program by prefixing them with EXEC SQL and suffixing the command with a semicolon (;). These statements are referred to as **static** statements.

Static statements can contain references to host variables, as described in the section."Using host variables" on page 20. All examples to this point have used static Embedded SQL statements.

Host variables can only be used in place of string or numeric constants. They cannot be used to substitute column names or table names; dynamic statements are required to do those operations.

Dynamic SQL statements

In the C language, strings are stored in arrays of characters. Dynamic statements are constructed in C language strings. These statements can then be executed using the PREPARE and EXECUTE statements. These SQL statements cannot reference host variables in the same manner as static statements since the C language variables are not accessible by name when the C program is executing.

To pass information between the statements and the C language variables, a data structure called the **SQL Descriptor Area** (**SQLDA**) is used. This structure is set up for you by the SQL preprocessor if you specify a list of host variables on the EXECUTE command in the USING clause. These variables correspond by position to place holders in the appropriate positions of the prepared command string.

For information on the SQLDA, see "The SQL descriptor area (SQLDA)" on page 45.

A **place holder** is put in the statement to indicate where host variables are to be accessed. A place holder is either a question mark (?) or a host variable reference as in static statements (a host variable name preceded by a colon). In the latter case, the host variable name used in the actual text of the statement serves only as a place holder indicating a reference to the SQL descriptor area.

A host variable used to pass information to the database is called a **bind variable**.

Example

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
   char address[30];
   char city[20];
   short int cityind;
   long empnum;
EXEC SQL END DECLARE SECTION;
. . .
   sprintf( comm, "update %s set address = :?,
                      city = :?"
            " where employee_number = :?",
            tablename );
EXEC SQL PREPARE S1 FROM :comm;
EXEC SQL EXECUTE S1 USING :address, :city:cityind,
:empnum;
```

This method requires the programmer to know how many host variables there are in the statement. Usually, this is not the case. So, you can set up your own SQLDA structure and specify this SQLDA in the USING clause on the EXECUTE command.

The DESCRIBE BIND VARIABLES statement returns the host variable names of the bind variables that are found in a prepared statement. This makes it easier for a C program to manage the host variables. The general method is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
. . .
sprintf( comm, "update %s set address = :address,
          city = :city"
         " where employee_number = :empnum",
         tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* Assume that there are no more than 10 host variables.
See next example if you can't put
a limit on it */
sqlda = alloc_sqlda( 10 );
```

```
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 USING DESCRIPTOR
sqlda;
/* sqlda->sqld will tell you how many host variables
there were. */
/* Fill in SQLDA_VARIABLE fields with values based on
name fields in sqlda */
. . .
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
free_sqlda( sqlda );
```

SQLDA contents

The SQLDA consists of an array of variable descriptors. Each descriptor describes the attributes of the corresponding C program variable, or the location that the database stores data into or retrieves data from:

- data type
- length if **type** is a string type
- precision and scale if **type** is a numeric type
- memory address
- indicator variable

For a complete description of the SQLDA structure, see "The SQL descriptor area (SQLDA)" on page 45

Indicator variables and NULL

The indicator variable is used to pass a NULL value to the database or retrieve a NULL value from the database. The indicator variable is also used by the database server to indicate truncation conditions encountered during a database operation. The indicator variable is set to a positive value when not enough space was provided to receive a database value.

For more information, see "Indicator variables" on page 23.

Dynamic SELECT statement

A SELECT statement that returns only a single row can be prepared dynamically, followed by an EXECUTE with an INTO clause to retrieve the one-row result. SELECT statements that return multiple rows, however, are managed using dynamic cursors.

With dynamic cursors, results are put into a host variable list or a SQLDA that is specified on the FETCH statement (FETCH INTO HOSTLIST and FETCH USING DESCRIPTOR SQLDA). Since the number of select list items is usually unknown to the C programmer, the SQLDA route is the most common. The DESCRIBE SELECT LIST statement sets up a SQLDA with the types of the select list items. Space is then allocated for the values using the **fill_sqlda()** function, and the information is retrieved by the FETCH USING DESCRIPTOR statement.

The typical scenario is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
   int actual_size;
   SQLDA * sqlda;

. . .
sprintf( comm, "select * from %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result. If it is
   wrong, it is corrected right after the first
   DESCRIBE by reallocating sqlda and doing DESCRIBE
   again. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1 USING DESCRIPTOR
sqlda;
if( sqlda->sqld > sqlda->sqln ){
   actual_size = sqlda->sqld;
   free_sqlda( sqlda );
   sqlda = alloc_sqlda( actual_size );
   EXEC SQL DESCRIBE SELECT LIST FOR S1
      USING DESCRIPTOR sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; ){
   EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
   if( SQLCODE == SQLE_NOTFOUND ) break;
   /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

Drop statements after use
You should ensure that statements are dropped after use, to avoid consuming unnecessary resources.

☞ For a complete example using cursors for a dynamic select statement , see "Dynamic cursor example" on page 75. For details of the functions mentioned above, see "Library functions" on page 55.

Fetching more than one row at a time

The FETCH statement can be modified to fetch more than one row at a time, which may improve performance. This is called a **wide fetch**.

Adaptive Server Anywhere also supports wide puts and inserts. For information on these, see "PUT statement" on page 524 of the book *Adaptive Server Anywhere Reference Manual* and "EXECUTE statement" on page 460 of the book *Adaptive Server Anywhere Reference Manual*.

To use wide fetches in Embedded SQL, include the fetch statement in your code as follows:

```
EXEC SQL FETCH . . . ARRAY nnn
```

where *ARRAY nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The number of variables in the SQLDA must be the product of *nnn* and the number of columns per row. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

The server returns in SQLCOUNT the number of records that were fetched, which is always greater than zero unless there is an error. A SQLCOUNT of zero with no error condition indicates that one valid row has been fetched.

Example

The following example code illustrates the use of wide fetches. The example code is not compilable as it stands.

```
EXEC SQL BEGIN DECLARE SECTION;
static unsigned             FetchWidth;
EXEC SQL END DECLARE SECTION;

static SQLDA * DoWideFetches( a_sql_statement_number
stat0,
                              unsigned
*num_of_rows,
                              unsigned
*cols_per_row )
/*******************************************************
**************/
// Allocate a SQLDA to be used for fetching from the
statement identified
// by "stat0".  "width" rows is retrieved on each FETCH
request.
// The number of columns retrieved per row is assigned
to "cols_per_row".
{
    int                     num_cols;
    unsigned                i, j, offset;
    SQLDA *                 sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    EXEC SQL END DECLARE SECTION;
```

```
    stat = stat0;
    sqlda = alloc_sqlda( 100 );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
    *cols_per_row = num_cols = sqlda->sqld;
    if( (num_cols * *num_of_rows) > sqlda->sqln ) {
        free_sqlda( sqlda );
        sqlda = alloc_sqlda( num_cols * width );
        if( sqlda == NULL ) return( NULL );
        EXEC SQL DESCRIBE :stat INTO sqlda;
    }
    sqlda->sqld = num_cols * *num_of_rows;
    offset = num_cols;
    for( i = 1; i < width; ++i ) {
        for( j = 0; j < num_cols; ++j, ++offset ) {
            sqlda->sqlvar[offset].sqltype = sqlda-
>sqlvar[j].sqltype;
            sqlda->sqlvar[offset].sqllen = sqlda-
>sqlvar[j].sqllen;
            memcpy( &sqlda->sqlvar[offset].sqlname,
                    &sqlda->sqlvar[j].sqlname,
                    sizeof( sqlda->sqlvar[0].sqlname )
);
        }
    }
    fill_sqlda( sqlda );
    return( sqlda );
}

long DoQuery( char * qry )
/************************/
{
    long                    rows;
    unsigned                cols_per_row;
    SQLDA *                 sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    static unsigned         num_of_rows;
    EXEC SQL END DECLARE SECTION;

    rows = 0L;
    FetchWidth = 20;

    EXEC SQL WHENEVER SQLERROR GOTO err;

    stmt = qry;
    EXEC SQL PREPARE :stat FROM :stmt;

    EXEC SQL DECLARE QCURSOR CURSOR FOR :stat FOR READ
ONLY;
```

```
    EXEC SQL OPEN QCURSOR;
    sqlda = DoWideFetches( stat, &num_of_rows,
&cols_per_row );
    if( sqlda == NULL ) {
        printf( "Maximum allowable fetch width
exceeded\n" );
        return( SQLE_NO_MEMORY );
    }

    for( ;; ) {
        EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda
ARRAY :FetchWidth;
        if (SQLCODE != SQLE_NOERROR) break;
    if( SQLCOUNT == 0 ) {
        rows += 1;
    } else {
        rows += SQLCOUNT;
    }
    }

    EXEC SQL CLOSE QCURSOR;
    EXEC SQL DROP STATEMENT :stat;
    free_sqlda( sqlda );

err:
    if (SQLCODE != SQLE_NOERROR) {
        printf( "Error detected\n" );
    }

    return (SQLCODE);
}
```

Notes on using wide fetches

- In Windows 3.x, the limit on the size of a SQLDA is 1450 columns, and the number of rows times the number of columns per row must be no more than 1450. Further, the SQLDA itself (not including data items) must fit in a single 64K segment, and **alloc_sqlda** will return NULL on an attempt to allocate a SQLDA that is too large.
- In the function **DoWideFetches**, the SQLDA memory is allocated using the **alloc_sqlda** function. This allows space for indicator variables, rather than using the **alloc_sqlda_noind** function.

  If fewer than the requested number of rows are fetched (at the end of the cursor, for example), the SQLDA items corresponding to the rows that were not fetched are returned as NULL by setting the indicator value. If no indicator variables are present, an error is generated (SQLE_NO_INDICATOR: no indicator variable for NULL result).

- If a row being fetched has been updated, generating a SQLE_ROW_UPDATED_WARNING warning, the fetch stops on the row that caused the warning. The values for all rows processed to that point (including the row that caused the warning) are returned. SQLCOUNT contains the number of rows that were fetched, including the row that caused the warning. All remaining SQLDA items are marked as NULL.
- If a row being fetched has been deleted or is locked, generating an SQLE_NO_CURRENT_ROW or SQLE_LOCKED error, SQLCOUNT contains the number of rows that were read prior to the error. This does not include the row that caused the error. The SQLDA does not contain values for the rows, since SQLDA values are not returned on errors. The SQLCOUNT value can be used to reposition the cursor, if necessary, to read the rows.

The SQL descriptor area (SQLDA)

The SQLDA (SQL Descriptor Area) is an interface structure that is used for dynamic SQL statements. The structure passes information regarding host variables and SELECT statement results to and from the database. The SQLDA is defined in the header file *sqlda.h*.

There are functions in the database interface library or DLL that you can use to manage SQLDAs. For descriptions, see "SQLDA management functions" on page 60.

When host variables are used with static SQL statements, the preprocessor constructs a SQLDA for those host variables. It is this SQLDA that is actually passed to and from the database server.

SQLDA fields and their meanings

The SQLDA fields have the following meanings:

Field	Description
sqldaid	An 8-byte character field that contains the string **SQLDA** as an identification of the SQLDA structure. This field helps in debugging, when you are looking at memory contents.
sqldabc	A long integer containing the length of the SQLDA structure.
sqln	The number of variable descriptors in the **sqlvar** array.
sqld	The number of variable descriptors which are valid (contain information describing a host variable). This field is set by the DESCRIBE statement, and sometimes by the programmer when supplying data to the database server.
sqlvar	An array of descriptors of type **struct sqlvar,** each describing a host variable.

Host variable descriptions in the SQLDA

Each **sqlvar** structure in the SQLDA describes a host variable. The fields of the **sqlvar** structure have the following meanings:

- **sqltype** The type of the variable that is described by this descriptor (see "Embedded SQL data types" on page 17).

  The low order bit indicates whether NULL values are allowed. Valid types and constant definitions can be found in the *sqldef.h* header file.

This field is filled by the DESCRIBE statement. You can set this field to any type, when supplying data to the database server or retrieving data from the database server. Any necessary type conversion is done automatically.

- **sqllen** The length of the variable. What the length actually means depends upon the type information and how the SQLDA is being used.

  For DECIMAL types, this field is divided into two 1-byte fields. The high byte is the precision and the low byte is the scale. The precision is the total number of digits. The scale is the number of digits that appear after the decimal point.

  For more information on the length field, see "Length field values" on page 47.

- **sqldata** A four-byte pointer to the memory occupied by this variable. This memory must correspond to the **sqltype** and **sqllen** fields.

  For storage formats, see "Embedded SQL data types" on page 17.

  For UPDATE and INSERT commands, this variable will not be involved in the operation if the **sqldata** pointer is a null pointer. For a FETCH, no data is returned if the **sqldata** pointer is a null pointer.

  If the DESCRIBE statement uses LONG NAMES, this field holds the long name of the result set column. If, in addition, the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type, instead of the column. If the type is a base type, the field is empty.

- **sqlind** A pointer to the indicator value. An indicator value is a **short int**. A negative indicator value indicates a NULL value. A positive indicator value indicates that this variable has been truncated by a FETCH statement, and the indicator value contains the length of the data before truncation.

  For more information, see "Indicator variables" on page 23.

  If the **sqlind** pointer is the null pointer, no indicator variable pertains to this host variable.

  The **sqlind** field is also used by the DESCRIBE statement to indicate parameter types. If the type is a user-defined data type, this field is set to DT_HAS_USERTYPE_INFO. In such a case, you may wish to carry out a DESCRIBE USER TYPES to obtain information on the user-defined data types.

- **sqlname** A VARCHAR structure that contains a length and character buffer. It is filled by a DESCRIBE statement and is not otherwise used. This field has a different meaning for the two formats of the DESCRIBE statement:

- **SELECT LIST** The name buffer is filled with the column heading of the corresponding item in the select list.
- **BIND VARIABLES** The name buffer is filled with the name of the host variable that was used as a bind variable, or "?" if an unnamed parameter marker is used.

On a DESCRIBE SELECT LIST command, any indicator variables present are filled with a flag indicating whether the select list item is updatable or not. More information on this flag can be found in the *sqldef.h* header file.

If the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type, instead of the column. If the type is a base type, the field is empty.

Length field values

The **sqllen** field length of the **sqlvar** structure in a SQLDA is used in three different kinds of interactions with the database server. The following tables detail each of these interactions. These tables list the interface constant types (the **DT_** types) found in the *sqldef.h* header file. These constants would be placed in the SQLDA **sqltype** field. The types are described in "Embedded SQL data types" on page 17.

In static SQL, a SQLDA is still used but it is generated and completely filled in by the SQL preprocessor. In this static case, the tables give the correspondence between the static C language host variable types and the interface constants.

DESCRIBE

The following table indicates the values of the **sqllen** and **sqltype** structure members returned by the DESCRIBE command for the various database types (both SELECT LIST and BIND VARIABLE DESCRIBE statements). In the case of a user-defined database data type, the base type is described.

Your program can use the types and lengths returned from a DESCRIBE, or you may use another type. The database server will perform type conversions between any two types. The memory pointed to by the **sqldata** field must correspond to the **sqltype** and **sqllen** fields.

Database field type	Embedded SQL type returned	Length returned on describe
CHAR(n)	DT_FIXCHAR	n
VARCHAR(n)	DT_VARCHAR	n
BINARY(n)	DT_BINARY	n
SMALLINT	DT_SMALLINT	2

Database field type	Embedded SQL type returned	Length returned on describe
INT	DT_INT	4
TINYINT	DT_TINYINT	1
DECIMAL(p,s)	DT_DECIMAL	high byte of length field in SQLDA set to p, and low byte set to s
REAL	DT_FLOAT	4
FLOAT	DT_FLOAT	4
DOUBLE	DT_DOUBLE	8
DATE	DT_DATE	length of longest formatted string
TIME	DT_TIME	length of longest formatted string
TIMESTAMP	DT_TIMESTAMP	length of longest formatted string
LONG VARCHAR	DT_VARCHAR	32767
LONG BINARY	DT_BINARY	32767

Supplying a value

The following table indicates how you specify lengths of values when you supply data to the database server in the SQLDA.

Only the data types shown in the table are allowed in this case. The DT_DATE, DT_TIME and DT_TIMESTAMP types are treated the same as DT_STRING when supplying information to the database; the value must be a NULL-terminated character string in an appropriate date format.

Embedded SQL Data Type	What the program must do to set the length when supplying data to the database
DT_STRING	length determined by terminating \0
DT_VARCHAR(n)	length taken from field in VARCHAR structure
DT_FIXCHAR(n)	length field in SQLDA determines length of string
DT_BINARY(n)	length taken from field in BINARY structure
DT_SMALLINT	No action required
DT_INT	No action required
DT_DECIMAL(p,s)	high byte of length field in SQLDA set to p,

Embedded SQL Data Type	What the program must do to set the length when supplying data to the database
	and low byte set to s
DT_FLOAT	No action required
DT_DOUBLE	No action required
DT_DATE	length determined by terminating \0
DT_TIME	length determined by terminating \0
DT_TIMESTAMP	length determined by terminating \0
DT_TIMESTAMP_STRUCT	No action required
DT_VARIABLE	length determined by terminating \0

Retrieving a value

The following table indicates the values of the length field when you retrieve data from the database using a SQLDA. The **sqllen** field is never modified when you retrieve data.

Only the interface data types shown in the table are allowed in this case. The DT_DATE, DT_TIME and DT_TIMESTAMP data types are treated the same as DT_STRING when you retrieve information from the database. The value is formatted as a character string in the current date format.

Embedded SQL Data Type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_STRING	length of buffer	\0 at end of string
DT_VARCHAR(n)	maximum length of VARCHAR structure (n+2)	`len` field of VARCHAR structure set to actual length
DT_FIXCHAR(n)	length of buffer	padded with blanks to length of buffer
DT_BINARY(n)	maximum length of BINARY structure (n+2)	`len` field of BINARY structure set to actual length
DT_SMALLINT	No action required	No action required
DT_INT	No action required	No action required
DT_DECIMAL(p,s)	high byte set to p and low byte set to s	No action required
DT_FLOAT	No action required	No action required

Embedded SQL Data Type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_DOUBLE	No action required	No action required
DT_DATE	length of buffer	\0 at end of string
DT_TIME	length of buffer	\0 at end of string
DT_TIMESTAMP	length of buffer	\0 at end of string
DT_TIMESTAMP_ STRUCT	No action required	No action required

Using stored procedures in Embedded SQL

This section describes the use of SQL procedures in Embedded SQL.

Simple procedures

Database procedures can be both created and called from Embedded SQL. A CREATE PROCEDURE statement can be embedded just like any other DDL statement. A CALL statement can also be embedded, or it can be prepared and executed. Here is a simple example of both creating and executing a stored procedure in Embedded SQL:

```
EXEC SQL CREATE PROCEDURE pettycash( IN amount
DECIMAL(10,2) )
BEGIN

   UPDATE account
   SET balance = balance - amount
   WHERE name = 'bank';

   UPDATE account
   SET balance = balance + amount
   WHERE name = 'pettycash expense';

END;
EXEC SQL CALL pettycash( 10.72 );
```

If you wish to pass host variable values to a stored procedure, or retrieve the output variables, you prepare and execute a CALL statement. The example illustrates the use of host variables. Both the USING and INTO clauses are used on the EXECUTE statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   double hv_expense;
   double hv_balance;
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE pettycash(
         IN expense   DECIMAL(10,2),
         OUT endbalance DECIMAL(10,2) )
   BEGIN
      UPDATE account
      SET balance = balance - expense
      WHERE name = 'bank';

      UPDATE account
      SET balance = balance + expense
      WHERE name = 'pettycash expense';
```

```
        SET endbalance = ( SELECT balance FROM account
                            WHERE name = 'bank' );
    END;

EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';

EXEC SQL EXECUTE S1 USING :hv_expense INTO :hv_balance;
```

Procedures with result sets

Database procedures can also contain SELECT statements. The procedure is declared using a RESULT clause to specify the number, name, and types of the columns in the result set. Result set columns are different from output parameters. For procedures with result sets, the CALL statement can be used in place of a SELECT statement in the cursor declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   char   hv_name[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE female_employees()
   RESULT( name char(50) )
   BEGIN
      SELECT emp_fname || emp_lname FROM employee
      WHERE sex = 'f';
   END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;) {
   EXEC SQL FETCH C1 INTO :hv_name;
   if( SQLCODE != SQLE_NOERROR ) break;
   printf( "%s\\n", hv_name );
}
EXEC SQL CLOSE C1;
```

In this example, the procedure has been invoked with an OPEN statement rather than an EXECUTE statement. The OPEN statement causes the procedure to execute until it reaches a SELECT statement. At this point, C1 is a cursor for the SELECT statement within the database procedure. You can use all forms of the FETCH command (backward and forward scrolling) until you are finished with it. The CLOSE statement terminates execution of the procedure.

If there had been another statement following the SELECT in the procedure, it would not have been executed. In order to execute statements following a SELECT, use the RESUME cursor-name command. The RESUME command will either return the warning SQLE_PROCEDURE_COMPLETE, or it will return SQLE_NOERROR indicating that there is another cursor. The example illustrates a two-select procedure:

```
EXEC SQL CREATE PROCEDURE people()
RESULT( name char(50) )
BEGIN

   SELECT emp_fname || emp_lname
   FROM employee;

   SELECT fname || lname
   FROM customer;
END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR ) {
   for(;;) {
      EXEC SQL FETCH C1 INTO :hv_name;
      if( SQLCODE != SQLE_NOERROR ) break;
      printf( "%s\\n", hv_name );
   }
   EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

Dynamic cursors for CALL statements

These examples have used static cursors. Full dynamic cursors can also be used for the CALL statement

For a description of dynamic cursors, see "Dynamic SELECT statement" on page 39.

The DESCRIBE statement works fully for procedure calls. A DESCRIBE OUTPUT produces a SQLDA that has a description for each of the result set columns.

If the procedure does not have a result set, the SQLDA has a description for each INOUT or OUT parameter for the procedure. A DESCRIBE INPUT statement will produce a SQLDA having a description for each IN or INOUT parameter for the procedure.

DESCRIBE ALL

DESCRIBE ALL describes IN, INOUT, OUT and RESULT set parameters. DESCRIBE ALL uses the indicator variables in the SQLDA to provide additional information.

The DT_PROCEDURE_IN and DT_PROCEDURE_OUT bits are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns have both bits clear.

After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE).

For a complete description, see "DESCRIBE statement" on page 446 of the book *Adaptive Server Anywhere Reference Manual*.

Multiple result sets

If you have a procedure that returns multiple result sets, you must redescribe after each RESUME statement if the result sets change shapes.

You need to describe the cursor, not the statement number, to describe the current position of the cursor.

Library functions

The SQL preprocessor generates calls to functions in the interface library or DLL. In addition to the calls generated by the SQL preprocessor, several routines are provided for the user to make database operations easier to perform. Prototypes for these functions are included by the EXEC SQL INCLUDE SQLCA command.

This section contains a detailed description of these various functions by category.

DLL entry points

The DLL entry points are the same except that the prototypes have a modifier appropriate for DLLs:

- **Windows 3.x:** FAR PASCAL
- **Windows NT:** __stdcall

All of the pointers that are passed as parameters to the Windows DLL entry points or returned by these functions are far pointers.

Example

For example, the first prototype listed below is **db_init**. For Windows, it would be:

```
unsigned short FAR PASCAL db_init( struct sqlca far *sqlca );
```

Passing null pointers
Care should be taken passing the null pointer as a parameter in Windows if your program is compiled in the small or medium memory models. You should use the **_sql_ptrchk_()** macro defined in *sqlca.h* for any pointer parameter which is a variable that might contain the null pointer. This macro converts a null near pointer into a null far pointer.

Interface initialization functions

This section lists the functions that initialize and release the interface.

db_init function

Prototype

```
unsigned short db_init( struct sqlca *sqlca );
```

Description

This function initializes the database interface library. This function must be called before any other library call is made, and before any Embedded SQL command is executed. The resources the interface library requires for your program are allocated and initialized on this call.

Use **db_fini** to free the resources at the end of your program. If there are any errors during processing, they are returned in the SQLCA and 0 is returned. If there are no errors, a non-zero value is returned and you can begin using Embedded SQL commands and functions.

In most cases, this function should be called only once (passing the address of the global **sqlca** variable defined in the *sqlca.h* header file). If you are writing a DLL or an application that has multiple threads using Embedded SQL, call **db_init** once for each SQLCA that is being used (see "SQLCA management for multi-threaded or reentrant code" on page 29).

> ***Caution***
> *Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail, and the NetWare file server to fail.*

db_fini function

Prototype

```
unsigned short db_fini( struct sqlca *sqlca );
```

This function frees resources used by the database interface or DLL. You must not make any other library calls or execute any Embedded SQL commands after **db_fini** is called. If there are any errors during processing, they are returned in the SQLCA and 0 is returned. If there are no errors, a non-zero value is returned.

You need to call **db_fini** once for each SQLCA being used.

> ***Caution***
> *Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail, and the NetWare file server to fail.*

Connection and server management functions

The following functions provide a means to start and stop the database server or, start or stop a database on an existing database server; and connect to or disconnect from a database.

All of these functions take a NULL-terminated string as the second argument. This string is a list of parameter settings of the form KEYWORD=**value**, delimited by semicolons. The number sign (#) is an alternative to the equals sign, and should be used when the equals sign is a syntax error, such as in environment variables on some platforms..

☞ For an available list of connection parameters, see "Connection parameters" on page 40 of the book *Adaptive Server Anywhere Reference Manual*.

Each function uses a subset of the available connection parameters, but every function will allow any parameter to be set. A sample connection parameter string is:

```
"UID=dba;PWD=sql;DBF=c:\asa6\asademo.db"
```

When included in Embedded SQL, the backslash character (\) must be escaped with a second backslash character in order to work:

```
"UID=dba;PWD=sql;DBF=c:\\asa6\\asademo.db"
```

db_string_connect function

Prototype unsigned db_string_connect(struct sqlca * sqlca, char * parms);

Description Provides extra functionality beyond the Embedded SQL CONNECT command. This function carries out the following steps:

- Start the database server if there is not one running with the name **EngineName** (calls **db_start_engine**). The **AutoStop** parameter determines if the server automatically stops when the last used database is shut down.
- If the database named by **DatabaseName** or **DatabaseFile** is not currently running, send a request to the server to start a database using the **DatabaseFile**, **DatabaseName**, and **DatabaseSwitches** parameters. The **AutoStop** parameter determines if the database automatically shuts down when the last connection to the database is disconnected.
- Send a connection request to the database server based on the **Userid**, **Password**, and **ConnectionName** parameters.

The return value is true (non-zero) if a connection was successfully established and false (zero) otherwise. Error information for starting the server, starting the database, or connecting is returned in the SQLCA.

db_string_disconnect function

Prototype unsigned db_string_disconnect(struct sqlca * sqlca, char * parms);

Description This function disconnects the connection identified by the **ConnectionName** parameter. All other parameters are ignored.

If no **ConnectionName** parameter is specified in the string, the unnamed connection is disconnected. This is equivalent to the Embedded SQL DISCONNECT command. The boolean return value is true if a connection was successfully ended. Error information is returned in the SQLCA.

This function shuts down the database if it was started with the **AutoStop**=**yes** parameter and there are no other connections to the database. It also stops the server if it was started with the **AutoStop**=**yes** parameter and there are no other databases running.

db_start_engine function

Prototype

unsigned db_start_engine(struct sqlca * sqlca,
    char * parms);

Description

Start the database server if it is not running. The steps carried out by this function are those listed in "Starting a personal server" on page 55 of the book *Adaptive Server Anywhere User's Guide*.

The return value is true if a database server was either found or successfully started. Error information is returned in the SQLCA.

The following call to **db_start_engine** starts the database server and names it **asademo**, but does not load the database, despite the DBF connection parameter:

```
db_start_engine( &sqlca, "DBF=c:\\asa6\\asademo.db;
Start=DBENG6" );
```

If you wish to start a database as well as the server, include the database file in the START connection parameter:

```
db_start_engine( &sqlca,"ENG=eng_name;START=DBENG6
c:\\asa6\\asademo.db" );
```

This call starts the server, names it **eng_name**, and starts the **asademo** database on that server.

db_start_database function

Prototype

unsigned db_start_database(struct sqlca * sqlca, char * parms);

Description

Start a database on an existing server if the database is not already running. The steps carried out to start a database are described in "Starting a personal server" on page 55 of the book *Adaptive Server Anywhere User's Guide*

The return value is true if the database was already running or successfully started. Error information is returned in the SQLCA.

If a user ID and password are supplied in the parameters, they are ignored.

☞ The permission required to start and stop a database is set on the database command line. For information, see "The database server" on page 12 of the book *Adaptive Server Anywhere Reference Manual*.

db_stop_database function

Prototype

```
unsigned int db_stop_database( struct sqlca * sqlca,
        char * parms );
```

Description

Stop the database identified by **DatabaseName** on the server identified by **EngineName**. If **EngineName** is not specified, the default server is used.

By default, this function does not stop a database that has existing connections. If **Unconditional** is **yes**, the database is stopped regardless of existing connections.

A return value of TRUE indicates that there were no errors.

☞ The permission required to start and stop a database is set on the database command line. For information, see "The database server" on page 12 of the book *Adaptive Server Anywhere Reference Manual*.

db_stop_engine function

Prototype

```
unsigned int db_stop_engine( struct sqlca * sqlca,
        char * parms );
```

Description

Terminates execution of the database server. The steps carried out by this function are:

- Look for a local database server that has a name that matches the **EngineName** parameter. If no **EngineName** is specified, look for the default local database server.
- If no matching server is found, this function fails.
- Send a request to the server to tell it to checkpoint and shut down all databases.
- Unload the database server.

By default, this function will not stop a database server that has existing connections. If **Unconditional** is *yes*, the database server is stopped regardless of existing connections.

A C program can use this function instead of spawning DBSTOP. A return value of TRUE indicates that there were no errors.

db_find_engine function

Prototype

unsigned short db_find_engine(struct sqlca *sqlca, char *name);

Description

Returns an unsigned short value, which indicates status information about the database server whose name is *name*. If no server can be found with the specified name, the return value is 0. A non-zero value indicates that the server is currently running.

Each bit in the return value conveys some information. Constants that represent the bits for the various pieces of information are defined in the *sqldef.h* header file. If a null pointer is specified for *name*, information is returned about the default database environment.

SQLDA management functions

The following functions are used to manage SQL Descriptor Areas (SQLDAs).

For a detailed description of the SQLDA, see "The SQL descriptor area (SQLDA)" on page 45.

alloc_sqlda_noind function

Prototype

struct sqlda *alloc_sqlda_noind(unsigned numvar);

Description

Allocates a SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA is initialized to *numvar*. Space is not allocated for indicator variables; the indicator pointers are set to the null pointer. A null pointer is returned if memory cannot be allocated.

alloc_sqlda function

Prototype

struct sqlda *alloc_sqlda(unsigned numvar);

Description

Allocates a SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA is initialized to *numvar*. Space is allocated for the indicator variables, the indicator pointers are set to point to this space, and the indicator value is initialized to zero. A null pointer is returned if memory cannot be allocated.

fill_sqlda function

Prototype

struct sqlda *fill_sqlda(struct sqlda *sqlda);

Description

Allocates space for each variable described in each descriptor of *sqlda*, and assigns the address of this memory to the **sqldata** field of the corresponding descriptor. Enough space is allocated for the database type and length indicated in the descriptor. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

sqlda_string_length function

Prototype

unsigned long sqlda_string_length(struct sqlda *sqlda, int varno);

Description

Returns the length of the C string (type DT_STRING) that would be required to hold the variable **sqlda->sqlvar[varno]** (no matter what its type is).

sqlda_storage function

Prototype

unsigned long sqlda_storage(struct sqlda *sqlda, int varno);

Description

Returns the amount of storage required to store any value for the variable described in **sqlda->sqlvar[varno]**.

fill_s_sqlda function

Prototype

struct sqlda *fill_s_sqlda(struct sqlda *sqlda, unsigned int maxlen);

Description

Much the same as **fill_sqlda,** except that it changes all the data types in *sqlda* to type DT_STRING.. Enough space is allocated to hold the string representation of the type originally specified by the SQLDA, up to a maximum of *maxlen* bytes. The length fields in the SQLDA (**sqllen**) are modified appropriately. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

free_filled_sqlda function

Prototype

void free_filled_sqlda(struct sqlda *sqlda);

Description

Free the memory allocated to each **sqldata** pointer. Any null pointer is not freed. The indicator variable space, as allocated in **fill_sqlda,** is also freed.

free_sqlda_noind function

Prototype

void free_sqlda_noind(struct sqlda *sqlda);

Description

Free space that was allocated to this *sqlda*. You should first call **free_filled_sqlda** to free the memory referenced by each **sqldata** pointer. The indicator variable pointers are ignored.

free_sqlda function

Prototype

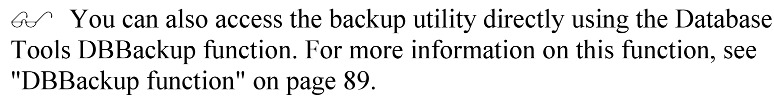
```
void free_sqlda( struct sqlda *sqlda );
```

Description

Free the space allocated to this *sqlda*. You should first call **free_filled_sqlda** to free the memory referenced by each **sqldata** pointer. The indicator variable space, as allocated in **fill_sqlda**, is also freed.

Backup functions

The **db_backup** function provides support for online backup. The Adaptive Server Anywhere backup utility makes use of this function. You should only need to write a program to use this function if your backup requirements are not satisfied by the Adaptive Server Anywhere backup utility.



You can also access the backup utility directly using the Database Tools DBBackup function. For more information on this function, see "DBBackup function" on page 89.

Every database contains one or more files. Normally, a database contains two files: the main database file and the transaction log.

Each file is divided into fixed size pages, and the size of these pages is specified when the database is created.

Backup works by opening a file, and then making a copy of each page in the file. Backup performs a checkpoint on startup, and the database files are backed up as of this checkpoint. Any changes that are made while the backup is running are recorded in the transaction log, and are backed up with the transaction log. This is why the transaction log is always backed up last.

Authorization

You must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote) to use the backup functions.

db_backup function

Prototype

```
void db_backup( struct sqlca * sqlca, int op, int file_num, unsigned long
      page_num, struct sqlda * sqlda);
```

Authorization

Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

Description

The action performed depends on the value of the *op* parameter:

- **DB_BACKUP_START** Must be called before a backup can start. Only one backup can be running at one time against any given database server. Database checkpoints are disabled until the backup is complete (**db_backup** is called with an *op* value of DB_BACKUP_END). If the backup cannot start, the SQLCODE is SQLE_BACKUP_NOT_STARTED. Otherwise, the SQLCOUNT field of the *sqlca* is set to the size of each database page. (Backups are processed one page at a time.)

  The *file_num*, *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_OPEN_FILE** Open the database file specified by *file_num*, which allows pages of the specified file to be backed up using DB_BACKUP_READ_PAGE. Valid file numbers are 0 through DB_BACKUP_MAX_FILE for the main database files, DB_BACKUP_TRANS_LOG_FILE for the transaction log file, and DB_BACKUP_WRITE_FILE for the database write file if it exists. If the specified file does not exist, the SQLCODE is SQLE_NOTFOUND. Otherwise, SQLCOUNT contains the number of pages in the file, SQLIOESTIMATE contains a 32-bit value (POSIX **time_t**) which identifies the time that the database file was created, and the operating system file name is in the *sqlerrmc* field of the SQLCA.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_READ_PAGE** Read one page of the database file specified by *file_num*. The *page_num* should be a value from 0 to one less than the number of pages returned in SQLCOUNT by a successful call to **db_backup** with the DB_BACKUP_OPEN_FILE operation. Otherwise, SQLCODE is set to SQLE_NOTFOUND. The *sqlda* descriptor should be set up with one variable of type DT_BINARY pointing to a buffer. The buffer should be large enough to hold binary data of the size returned in the SQLCOUNT field on the call to **db_backup** with the DB_BACKUP_START operation.

  DT_BINARY data contains a two-byte length followed by the actual binary data, so the buffer must be two bytes longer than the page size.

  Application must save buffer
  This call makes a copy of the specified database page into the buffer, but it is up to the application to save the buffer on some backup media.

- **DB_BACKUP_READ_RENAME_LOG** This action is the same as DB_BACKUP_READ_PAGE, except that after the last page of the transaction log has been returned, the database server renames the transaction log and starts a new one.

If the database server is unable to rename the log at the current time (there are incomplete transactions), you will get the SQLE_BACKUP_CANNOT_RENAME_LOG_YET error. In this case, don't use the page returned, but instead reissue the request until you receive SQLE_NOERROR and then write the page. Continue reading the pages until you receive the SQLE_NOTFOUND condition.

The SQLE_BACKUP_CANNOT_RENAME_LOG_YET error may be returned multiple times and on multiple pages. In your retry loop, you should add a delay so as not to slow the server down with too many requests.

When you receive the SQLE_NOTFOUND condition, the transaction log has been backed up successfully and the file has been renamed. The name for the old transaction file is returned in the *sqlerrmc* field of the SQLCA.

You should check the **sqlda->sqlvar[0].sqlind** value after a **db_backup** call. If this value is greater than zero, the last log page has been written and the log file has been renamed. The new name is still in **sqlca.sqlerrmc**, but the SQLCODE value is SQLE_NOERROR.

You should not call **db_backup** again after this. If you do, you get a second copy of your backed up log file and you receive SQLE_NOTFOUND.

- **DB_BACKUP_CLOSE_FILE** Must be called when processing of one file is complete to close the database file specified by *file_num*.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_END** Must be called at the end of the backup. No other backup can start until this backup has ended. Checkpoints are enabled again.

  The *file_num*, *page_num* and *sqlda* parameters are ignored.

The *dbbackup* program uses the following algorithm. Note that this is *not* C code, and does not include error checking.

```
db_backup( ... DB_BACKUP_START ... )
allocate page buffer based on page size in SQLCODE
sqlda = alloc_sqlda( 1 )
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_BINARY
sqlda->sqlvar[0].sqldata = allocated buffer
for file_num = 0 to DB_BACKUP_MAX_FILE
  db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
  if SQLCODE == SQLE_NO_ERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLE_IO_ESTIMATE
```

```
    open backup file with name from sqlca.sqlerrmc
    for page_num = 0 to num_pages - 1
      db_backup( ... DB_BACKUP_READ_PAGE,
                 file_num, page_num, sqlda )
      write page buffer out to backup file
    next page_num
    close backup file
    db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
  end if
next file_num
backup up file DB_BACKUP_WRITE_FILE as above
backup up file DB_BACKUP_TRANS_LOG_FILE as above
free page buffer
db_backup( ... DB_BACKUP_END ... )
```

db_delete_file function

Prototype

void db_delete_file(struct sqlca * sqlca,
    char * filename);

Authorization

Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

Description

The **db_delete_file** function requests the database server to delete *filename*. This can be used after backing up and renaming the transaction log (see DB_BACKUP_READ_RENAME_LOG above) to delete the old transaction log. You must be connected to a user ID with DBA authority.

Canceling a request

The following functions provide the ability to check whether a request is being processed, and to cancel a request.

db_cancel_request function

Prototype

int db_cancel_request(struct sqlca *sqlca);

Description

Cancels the currently active database server request. This function will check to make sure a database server request is active before sending the cancel request. The return value indicates whether a cancel request was sent; in other words, whether or not a database request was active.

A non-zero return value means that the request was not canceled. There are a few critical timing cases where the cancel request and the response from the database or server "cross". In these cases, the cancel simply has no effect.

The **db_cancel_request** function can be called asynchronously. This function and **db_is_working** are the only functions in the database interface library that can be called asynchronously using an SQLCA that might be in use by another request.

If you cancel a request that is carrying out a cursor operation, the position of the cursor is indeterminate. You must locate the cursor by its absolute position, or close it, following the cancel.

db_is_working function

Prototype

```
unsigned db_is_working( struct sqlca *sqlca );
```

Description

Returns 1 if your application has a database request in progress that uses the given sqlca, and 0 if there is no request in progress that uses the given sqlca.

This function can be called asynchronously. This function and **db_cancel_request** are the only functions in the database interface library that can be called asynchronously using an SQLCA that might be in use by another request.

Other functions

The following functions are miscellaneous functions.

sql_needs_quotes function

Prototype

```
unsigned int sql_needs_quotes( struct sqlca *sqlca, char *str );
```

Description

Returns a Boolean value that indicates whether the string requires double quotes around it when it is used as a SQL identifier. This function formulates a request to the database server to determine if quotes are needed. Relevant information is stored in the **sqlcode** field.

There are three cases of return value/code combinations:

- **return = FALSE, sqlcode = 0** In this case, the string definitely does not need quotes
- **return = TRUE** In this case, sqlcode is always SQLE_WARNING, and the string definitely does need quotes
- **return = FALSE** If sqlcode is something other than SQLE_WARNING, the test is inconclusive

sqlerror_message function

Prototype char *sqlerror_message(struct sqlca *sqlca, char * buffer, int max);

Description Return a pointer to a string that contains an error message. The error message contains text for the error code in the SQLCA. If no error was indicated, a null pointer is returned. The error message is placed in the buffer supplied, truncated to length *max* if necessary.

Request management functions

The default behavior of the interface DLL is for applications to wait for completion of each database request before carrying out other functions. This behavior can be changed using request management functions. For example, when using Interactive SQL, the operating system is still active while Interactive SQL is waiting for a response from the database, and Interactive SQL carries out some tasks in that time.

You can achieve application activity while a database request is in progress by providing a **callback function**. In this callback function you must not do another database request (except **db_cancel_request**). You can use the **db_is_working** function in your message handlers to determine if you have a database request in progress.

This callback function in your application is called repeatedly while the database server is busy processing a request. You can then process Windows messages by calling **GetMessage** or **PeekMessage**. (These function calls allow Windows to be active.) The *dblib6w.dll* continually calls this function until the response from the database server is received.

Response in Windows message
The response from the server comes to your application via a Windows message. You must either dispatch this message (the interface DLL will receive the message), or call the **db_process_message** function with each message that you receive while in this callback function. The function returns TRUE if the message was the response; otherwise you can process the message normally by dispatching it.

The following two functions are used to register your application callback functions:

db_register_a_callback function

Prototype

```
void db_register_a_callback( struct sqlca *sqlca,
                             a_db_callback_index index,
                             FARPROC callback );
```

Description

This function is used to register callback functions. To remove a callback, pass a null pointer as the *callback* function. You should call **MakeProcInstance** with your function address and pass that to the **db_register_a_callback** function.

If you do not register any callback functions, the default action is to do nothing. Your application blocks, waiting for the database response, and Windows changes the cursor to an hourglass.

The following values are allowed for the *index* parameter:

♦ **DB_CALLBACK_START** The prototype is as follows:

```
void FAR PASCAL db_start_request( struct sqlca
*sqlca );
```

This function is called just before a database request is sent to the server.

♦ **DB_CALLBACK_FINISH** The prototype is as follows:

```
void FAR PASCAL db_finish_request( struct sqlca
*sqlca );
```

This function is called after the response to a database request has been received by the interface DLL.

♦ **DB_CALLBACK_WAIT** The prototype is as follows:

```
void FAR PASCAL db_wait_request( struct sqlca *sqlca
);
```

This function is called repeatedly by the interface DLL while the database server or client library is busy processing your database request.

The following is a sample DB_CALLBACK_WAIT callback function:

```
void FAR PASCAL db_wait_request( struct sqlca *sqlca )
{ MSG msg
  if( GetMessage( &msg, NULL, 0, 0 ) ) {
    ( !db_process_a_message( sqlca, &msg ) ) {
      if( !TranslateAccelerator( hWnd, hAccel, &msg ) )
{
      TranslateMessage( &msg )
      DispatchMessage( &msg )
      }
    }
  }
}
```

♦ **DB_CALLBACK_MESSAGE** This is used to enable the application to handle messages received from the server during the processing of a request.

The callback prototype is as follows:

```
void FAR PASCAL message_callback( SQLCA* sqlca,
                  unsigned short msg_type,
                  an_sql_code code,
                  unsigned length,
                  char*  msg )
```

The **msg_type** parameter states how important the message is, and you may wish to handle different message types in different ways. The available message types are MESSAGE_TYPE_INFO, MESSAGE_TYPE_WARNING, MESSAGE_TYPE_ACTION and MESSAGE_TYPE_STATUS. These constants are defined in *sqldef.h*. The **code** field is an identifier. The **length** field tells you how long the message is. The message is *not* null-terminated, since it might be right in the data stream that we got from the server.

For example, the Interactive SQL callback displays STATUS and INFO message in the message window, while messages of type ACTION and WARNING go to a dialog box. If an application does not register this callback, there is a default callback, which causes all messages to be written to the server logfile (if debugging is on and a logfile is specified). In addition, messages of type MESSAGE_TYPE_WARNING and MESSAGE_TYPE_ACTION are more prominently displayed, in an operating system-dependent manner.

db_process_a_message function

Prototype int db_process_a_message(struct sqlca *sqlca, MSG *msg);

Description This function is called from within your **db_wait_request** callback function to determine if the message that you received from Windows was in fact the response to the active database request. The return value is TRUE if *msg* is the response. Return from the callback function and the Embedded SQL library DLL will process the response by returning to the call that generated the original request.

Embedded SQL commands

> **EXEC SQL**
> ALL Embedded SQL statements must be preceded with EXEC SQL and end with a semicolon.

There are two groups of Embedded SQL commands:

Standard SQL commands are used by simply placing them in a C program enclosed with EXEC SQL and a semi-colon. CONNECT, DELETE, SELECT, SET and UPDATE have additional formats only available in Embedded SQL. The additional formats fall into the second category of Embedded SQL specific commands.

All commands are described in detail in "SQL Statements" on page 339 of the book *Adaptive Server Anywhere Reference Manual*.

Several SQL commands are specific to Embedded SQL and can only be used in a C program.

These Embedded SQL commands are also described in "SQL Language Elements" on page 179 of the book *Adaptive Server Anywhere Reference Manual*.

The Embedded SQL commands include the following:

- **ALLOCATE DESCRIPTOR** Allocate memory for a descriptor
- **CLOSE** close a cursor
- **CONNECT** connect to the database
- **DEALLOCATE DESCRIPTOR** Reclaim memory for a descriptor
- **Declaration Section** declare host variables for database communication
- **DECLARE** declare a cursor
- **DELETE (positioned)** delete the row at the current position in a cursor
- **DESCRIBE** describe the host variables for a particular SQL statement
- **DISCONNECT** disconnect from database server
- **DROP STATEMENT** free resources used by a prepared statement
- **EXECUTE** execute a particular SQL statement
- **EXPLAIN** explain the optimization strategy for a particular cursor
- **FETCH** fetch a row from a cursor

- **GET OPTION** get the setting for a particular database option
- **INCLUDE** include a file for SQL preprocessing
- **OPEN** open a cursor
- **PREPARE** prepare a particular SQL statement
- **PUT** insert a row into a cursor
- **SET CONNECTION** change active connection
- **SET OPTION** change a database option value
- **SET SQLCA** use an SQLCA other than the default global one
- **UPDATE (positioned)** update the row at the current location of a cursor
- **WHENEVER** specify actions to occur on errors in SQL statements

Database examples

Two Embedded SQL examples are included with the Adaptive Server Anywhere installation. The static cursor Embedded SQL example, *cur.sqc*, demonstrates the use of static SQL statements. The dynamic cursor Embedded SQL example, *dcur.sqc*, demonstrates the use of dynamic SQL statements. In addition to these examples, you may find other programs and source files as part of the installation of Adaptive Server Anywhere which demonstrate features available for particular platforms.

File locations

Source code for the examples is installed as part of the Adaptive Server Anywhere installation. They are placed in the *cxmp* subdirectory of your Adaptive Server Anywhere installation directory.

Building the examples

Along with the sample program is a batch file, *makeall.bat*, that can be used to compile the sample program for the various environments and compilers supported by Adaptive Server Anywhere. For UNIX, use the shell script makeall.

The format of the command is as follows:

```
makeall {Example} {Platform} {Compiler}
```

The first parameter is the name of the example program that you want to compile. It is one of:

- **cur** static cursor example
- **dcur** dynamic cursor example

The second parameter is the platform in which the program is run. The platform can be one of the following:

- **WINDOWS** compile for 16-bit Windows.
- **WIN32** compile for 32-bit Windows 3.x using the Watcom 32-bit support.
- **WINNT** compile for Windows NT.
- **NETWARE** compile for Netware NLM.

The third parameter is the compiler to use to compile the program. The compiler can be one of:

- **WC** use Watcom C/C++
- **MC** use Microsoft C

- **BC** use Borland C
- **CS** use IBM C Set++ or Visual Age

Running the example programs

Each example program presents a console-type user interface where it prompts you for a command. The various commands manipulate a database cursor and print the query results on the screen. Simply type the letter of the command you wish to perform. Some systems may require you to press ENTER after the letter.

The commands are similar to the following, depending on which program you run:

- **p** print current page
- **u** move up a page
- **d** move down a page
- **b** move to the bottom of the page
- **t** move to the top of the page
- **i** insert a row (*dcur* only)
- **n** new table (*dcur* only)
- **q** quit
- **h** help (this list of commands)

Windows and Windows NT examples

The Windows versions of the example programs are real Windows programs. However, to keep the user interface code relatively simple, some simplifications have been made. In particular, these applications do not repaint their Windows on WM_PAINT messages except to reprint the prompt.

To run these programs, execute them using the *curwin* executable name:

Static cursor example

This example demonstrates the use of cursors. The particular cursor used here retrieves certain information from the **employee** table in the sample database. The cursor is declared statically, meaning that the actual SQL statement to retrieve the information is "hard coded" into the source program. This is a good starting point for learning how cursors work. The next example ("Dynamic cursor example" on page 75) takes this first example and converts it to use dynamic SQL statements.

For information on where the source code can be found and how to build this example program , see "Database examples" on page 72.

The C program with the Embedded SQL is shown below. The program looks much like a standard C program except there are Embedded SQL instructions that begin with EXEC SQL.

To reduce the amount of code that is duplicated by the **cur** and **dcur** example programs (and the **odbc** example), the mainlines and the data printing functions have been placed into a separate file. This is *mainch.c* for character mode systems, and *mainwin.c* for windowing environments.

The example programs each supply the following three routines, which are called from the mainlines.

- **WSQLEX_Init** Connects to the database and opens the cursor
- **WSQLEX_Process_Command** Processes commands from the user, manipulating the cursor as necessary.
- **WSQLEX_Finish** Closes the cursor and disconnect from the database.

The function of the mainline is to:

1. Call the WSQLEX_Init routine
2. Loop, getting commands from the user and calling WSQL_Process_Command until the user quits
3. Call the WSQLEX_Finish routine

Connecting to the database is accomplished with the Embedded SQL CONNECT command supplying the appropriate user ID and password.

The **open_cursor** routine both declares a cursor for the specific SQL command and also opens the cursor.

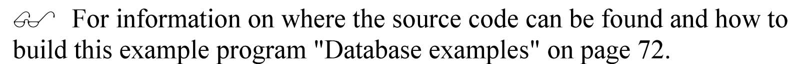
Printing a page of information is accomplished by the **print** routine. It loops *pagesize* times fetching a single row from the cursor and printing it out. Note that the fetch routine checks for warning conditions (such as End of Cursor) and prints appropriate messages when they arise. Also, the cursor is repositioned by this program to the row before the one that is displayed at the top of the current page of data.

The **move**, **top** and **bottom** routines use the appropriate form of the FETCH statement to position the cursor. Note that this form of the FETCH statement doesn't actually get the data — it only positions the cursor. Also, a general relative positioning routine, **move,** has been implemented to move in either direction depending on the sign of the parameter.

When the user quits, the cursor is closed and the database connection is also released. The cursor is closed by a ROLLBACK WORK statement, and the connection is release by a DISCONNECT.

Dynamic cursor example

This example demonstrates the use of cursors for a dynamic SQL SELECT statement. It is a slight modification of the previous example. If you have not yet looked at "Static cursor example" on page 74 it would be helpful to do so before looking at this example.


For information on where the source code can be found and how to build this example program "Database examples" on page 72.

The **dcur** program allows the user to select a table to look at with the **n** command. The program then presents as much information from that table as will fit on the screen. The SELECT statement is built up in a program array using the C library function **rintf**.

When this program is run, it prompts for a connection string of the form:

```
uid=dba;pwd=sql;dbf=c:\asa6\asademo.db
```

The C program with the Embedded SQL is shown below. The program looks much like the previous example with the exception of the **connect**, **open_cursor** and **print** functions.

The **connect** function uses the Embedded SQL interface function **db_string_connect** to connect to the database. This function provides the extra functionality to support the connection string that is used to connect to the database.

The **open_cursor** routine first builds the SELECT statement:

```
SELECT * FROM tablename
```

where *tablename* is a parameter passed to the routine. It then prepares a dynamic SQL statement using this string.

The Embedded SQL DESCRIBE command is used to fill in the SQLDA structure the results of the SELECT statement.

Size of the SQLDA
An initial guess is taken for the size of the SQLDA (3). If this is not big enough, the actual size of the select list returned by the database server is used to allocate a SQLDA of the right size.

The SQLDA structure is then filled with buffers to hold strings that represent the results of the query. The **fill_s_sqlda** routine converts all data types in the SQLDA to DT_STRING, and allocates buffers of the appropriate size.

A cursor is then declared and opened for this statement. The rest of the routines for moving and closing the cursor remain the same.

The **fetch** routine is slightly different: it puts the results into the SQLDA structure instead of into a list of host variables. The **print** routine has changed significantly to print results from the SQLDA structure up to the width of the screen. The **print** routine also uses the name fields of the SQLDA to print headings for each column.

Windows NT Service examples

The example programs *cur.sqc* and *dcur.sqc*, when compiled for Windows NT, run optionally as services.

The two files containing the example code for NT services are the source file *ntsvc.c* and the header file *ntsvc.h*. The code allows a linked executable to be run either as a regular executable or as an NT service.

❖ **To run one of the compiled examples as an NT service:**

1. Start Sybase Central, and open the Services folder.
2. Select a service type of Sample Application, and click OK.
3. Enter a service name in the appropriate field.
4. Select the sample program (*curwnt.exe* or *dcurwnt.exe*) from the *cxmp* subdirectory of the installation directory.
5. Click OK to install the service.
6. Click Start on the main window to start the service.

When run as a service, the programs display the normal user interface if possible. They also write the output to the Application Event Log. If it is not possible to start the user interface, the programs print one page of data to the Application Event Log and stop.

These examples have been tested with the Watcom C/C++ 10.5 compiler and the Microsoft Visual C++ 2.0 compiler.