CHAPTER 4

ODBC Programming

About this chapter

The **Open Database Connectivity** (**ODBC**) interface is a programming language interface defined by Microsoft Corporation as a standard interface to database management systems in the Windows and Windows NT environments.

This chapter presents information for those who use the ODBC interface directly. Users of application development systems that already have ODBC support do not need to read this chapter.

The primary documentation for ODBC application development is the ODBC SDK, available from Microsoft. This chapter describes features specific to Adaptive Server Anywhere, but is not an exhaustive guide to ODBC application programming.

Contents

Introduction to ODBC programming

The ODBC interface is defined by a set of function calls, called the ODBC API (Application Programming Interface).

To write ODBC applications for Adaptive Server Anywhere, you need:

- The Adaptive Server Anywhere software.
- A C compiler capable of creating programs for your environment.
- We also recommend that you obtain the Microsoft ODBC Software Development Kit. This is available on the Microsoft Developer Network, and provides documentation and additional tools for testing ODBC applications.

Applications that use the ODBC interface can work with many different database systems. Adaptive Server Anywhere supports the ODBC API on UNIX, in addition to the Windows and Windows NT environments. Having multi-platform ODBC support makes portable database application development much easier.

ODBC conformance

Adaptive Server Anywhere Version 6 provides support for ODBC 3.51.

Levels of ODBC support

ODBC features are arranged according to a level of conformance. Features are either **Core**, **Level 1**, or **Level 2**, with level 2 being the most complete level of ODBC support. These features are listed in the ODBC Programmer's Reference, which is available from Microsoft Corporation as part of the ODBC software development kit or from the Microsoft Web site, at the following Web site:

```
http://www.microsoft.com/data/odbc/techmat.htm.
```

Features supported by Adaptive Server Anywhere

Adaptive Server Anywhere supports the ODBC 3.51 specification.

- **Core conformance** Adaptive Server Anywhere supports all Core level features.
- **Level 1 conformance** Adaptive Server Anywhere supports all Level 1 features, except for asynchronous execution of ODBC functions.

  Adaptive Server Anywhere does support multiple threads sharing a single connection. The requests from the different threads are serialized by Adaptive Server Anywhere.
- **Level 2 conformance** Adaptive Server Anywhere supports all Level 2 features, except for the following:

- Three part names of tables and views. This is not applicable for Adaptive Server Anywhere.
- Asynchronous execution of ODBC functions for specified individual statements.
- Ability to time out login request and SQL queries.

ODBC backwards compatibility

Applications developed using older versions of ODBC will continue to work with Adaptive Server Anywhere and the newer ODBC driver manager. The new ODBC features are not provided for older applications.

The ODBC driver manager

The ODBC driver manager is part of the ODBC software supplied with Adaptive Server Anywhere. The ODBC Version 3 driver manager has a new interface for configuring ODBC data sources.

ODBC fundamentals

Database access in ODBC is carried out using SQL statements passed as strings to ODBC functions.

The following fundamental objects are used for every ODBC program. Each object is referenced by a **handle**.

- **Environment** Every ODBC application must allocate exactly one environment using **SQLAllocEnv**, and must free it at the end of the application using **SQLFreeEnv**. An environment handle is of type **HENV**.
- **Connections** An application can have several connections associated with its environment. Each connection must be allocated using **SQLAllocConnect** and freed using **SQLFreeConnect**. Your application can connect to a data source using either **SQLConnect**, **SQLDriverConnect** or **SQLBrowseConnect**, and disconnect using **SQLDisconnect**. A connection handle is of type **HDBC**.
- **Statements** Each connection can have several statements, each allocated using **SQLAllocStmt** and freed using **SQLFreeStmt**. Statements are used both for cursor operations (fetching data) and for single statement execution (e.g. INSERT, UPDATE and DELETE). A statement handle is of type **HSTMT**.

All access to these objects is through function calls; the application cannot directly access any information about the object from its handle. In the Windows and Windows NT environments, all the function calls are described in full detail in the ODBC API help file, which is part of the ODBC SDK.

Compiling and linking an ODBC application

Every C source file using ODBC functions must include one of the following lines:

- **Windows 95 and NT** #include "ntodbc.h"
- **Windows 3.x** #include "winodbc.h"

These files all include the main ODBC include file *odbc.h*, which defines all of the functions, data types and constant definitions required to write an ODBC program. The file *odbc.h* and the environment-specific include files are installed in the *h* subdirectory of your Adaptive Server Anywhere installation directory.

Once your program has been compiled, you must link with the appropriate import library file to have access to the ODBC functions:

- **Windows 95 and NT** *wodbc32.lib*, which defines the entry points into the Driver Manager *odbc32.dll*. If you connect to a database in Windows NT, *odbc32.dll* will load the ODBC driver, *dbodbc6.dll*.
- **Windows 3.x** *wodbc.lib*, which defines the entry points into the Driver Manager *odbc.dll*. If you connect to a database in Windows 3.x, *odbc.dll* will load the Adaptive Server Anywhere ODBC driver, *dbodbc6w.dll*.
- **UNIX** *dbodbc.lib*, which defines the entry points into the Adaptive Server Anywhere ODBC driver, *dbodbc6.so.*

A first example

The following is a simple ODBC program:

```
{
   HENV  env;
   HDBC  dbc;
   HSTMT stmt;

   SQLAllocEnv( &env );
   SQLAllocConnect( env, &dbc );
   SQLConnect( dbc, "asademo", SQL_NTS,
               "dba", SQL_NTS, "sql", SQL_NTS );
   SQLSetConnectOption( dbc, SQL_AUTOCOMMIT, FALSE );
   SQLAllocStmt( dbc, &stmt );

   /* Delete all the order items for order 2015 */
   SQLExecDirect( stmt,
      "DELETE FROM sales_order_items WHERE id=2015",
       SQL_NTS );

   /* Use rollback to undo the delete */
   SQLTransact( env, dbc, SQL_ROLLBACK );
   SQLFreeStmt( stmt, SQL_DROP );
   SQLDisconnect( dbc );
   SQLFreeConnect( dbc );
   SQLFreeEnv( env );
}
```

Notes

- **SQL_NTS** Every string passed to ODBC has a corresponding length. If the length is unknown, you can pass SQL_NTS indicating that it is a **Null Terminated String** whose end is marked by the null character (\0).
- **SQLSetConnectOption** By default, ODBC operates in auto-commit mode. This mode is turned off by setting SQL_AUTOCOMMIT to false.

- **SQLExecDirect** Executes the SQL statement specified in the parameter.
- **SQLTransact** Used to perform COMMIT and ROLLBACK statements marking the end of a transaction. You should not use **SQLExecDirect** to perform COMMIT or ROLLBACK.

Threads in ODBC applications

You can develop multi-threaded ODBC applications for Adaptive Server Anywhere. It is recommended that you use a separate connection for each thread.

You can use a single connection for multiple threads. However, the database server does not allow more than one active request for any one connection at a time. If one thread executes a statement that takes a long time, all other threads must wait until the request is complete.

Error checking in ODBC

The previous example did not check for any errors. Errors in ODBC are reported using the return value from each of the ODBC API function calls and the **SQLError** function.

Every ODBC API function returns a **RETCODE**, which is one of the following status codes:

- **SQL_SUCCESS** No error.
- **SQL_SUCCESS_WITH_INFO** The function completed, but a call to **SQLError** will indicate a warning. The most common case for this status is that a value being returned is too long for the buffer provided by the application.
- **SQL_ERROR** The function did not complete due to an error. You can call **SQLError** to get more information on the problem.
- **SQL_INVALID_HANDLE** An invalid environment, connection, or statement handle was passed as a parameter. This often happens if a handle is used after it has been freed, or if the handle is the null pointer.
- **SQL_NO_DATA_FOUND** There is no information available. The most common use for this status is when fetching from a cursor; it indicates that there are no more rows in the cursor.
- **SQL_NEED_DATA** Data is needed for a parameter. This is an advanced feature described in the help file under **SQLParamData** and **SQLPutData**.

Every environment, connection, and statement handle can have one or more errors or warnings associated with it. Each call to **SQLError** returns the information for one error and removes the information for that error. If you do not call **SQLError** to remove all errors, the errors are removed on the next function call that passes the same handle as a parameter.

Example

The following program fragment uses **SQLError** and return codes:

```
HDBC dbc;
HSTMT stmt;
RETCODE retcode;
UCHAR errmsg[100];

. . .

retcode = SQLAllocStmt( dbc, &stmt );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
             errmsg, sizeof(errmsg), NULL );
```

```
   /* Assume that print_error is defined */
   print_error( "Failed SQLAllocStmt", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "delete from sales_order_items
      where id=2015", SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
            errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Failed to delete items", errmsg );
   return;
}
. . .
```

Note that each call to **SQLError** passes three handles for an environment, connection, and statement. The first call uses SQL_NULL_HSTMT to get the error associated with a connection. Similarly, a call with both SQL_NULL_DBC and SQL_NULL_HSTMT will get any error associated with the environment handle.

The return value from **SQLError** may seem confusing. It returns SQL_SUCCESS if there is an error to report (not SQL_ERROR), and SQL_NO_DATA_FOUND if there are no more errors to report.

The examples pass the null pointer for some of the parameters to **SQLError**. The help file contains a full description of **SQLError** and all its parameters.

Using prepared statements in ODBC

Prepared statements provide performance advantages for statements that are used repeatedly. ODBC provides a full set of functions for using prepared statements.

☞ For an introduction to prepared statements, see "Preparing statements" on page 202 of the book *Adaptive Server Anywhere User's Guide*.

❖ **To execute a prepared statement:**

1 You prepare the statement using **SQLPrepare**. The following code fragment illustrates how to prepare an INSERT statement:

```
SQLRETURN   retcode;
SQLHSTMT    hstmt;

retcode = SQLPrepare(hstmt,
            "INSERT
             INTO department
             (dept_id, dept_name, dept_head_id )
             VALUES (?, ?, ?,)",
          SQL_NTS);
```

In this example:

- **retcode** Holds a return code that should be tested for success or failure of the operation.
- **hstmt** Provides a handle to the statement, so that it can be referenced later.
- **?** The question marks are placeholders for statement parameters.

2 You set statement parameter values using **SQLBindParameter**. For example, the following function call sets the value of the **dept_id** variable:

```
SQLBindParameter(hstmt,
                 1,
                 SQL_PARAM_INPUT,
                 SQL_C_SSHORT,
                 SQL_INTEGER,
                 0,
                 0,
                 &sDeptID,
                 0,
                 &cbDeptID);
```

In this example:

- **hstmt** The statement handle

- 1 indicates that this call sets the value of the first placeholder.
- SQL_PARAM_INPUT indicates that the parameter is an input statement.
- SQL_C_SHORT and SQL_INTEGER indicate the C data type being used in the application and the SQL type being used in the database.
- The next two parameters indicate the column precision and the number of decimal digits: both zero for integers.
- **&sDeptID** Pointer to a buffer for the parameter value.
- The following zero indicates the length of the buffer, in bytes.
- **&cbDeptID** Pointer to a buffer for the length of the parameter value.

3 Bind the other two parameters and assign values to sDeptId:

4 Execute the statement:

```
retcode = SQLExecute(hstmt);
```

This step can be carried out multiple times.

5 Drop the statement. This frees resources associated with the statement itself. You drop statements using **SQLFreeStmt**.

For more information, see the ODBC SDK documentation.

Working with result sets

ODBC provides extensive support for different kinds of cursor and cursor operations.

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

A cursor is opened using **SQLExecute** or **SQLExecDirect**, rows are fetched using **SQLFetch** or **SQLExtendedFetch** and the cursor is closed using **SQLFreeStmt**.

To fetch values from a cursor, the application can use either **SQLBindCol** or **SQLGetData**. If you use **SQLBindCol**, values are automatically retrieved on each fetch. If you use **SQLGetData**, you must call it for each column after each fetch.

The following code fragment opens and reads a cursor. Error checking has been omitted to make the example easier to read.

```
. . .
HDBC dbc;
HSTMT stmt;
RETCODE retcode;
long emp_id;
char emp_lname[20];

SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
             "select emp_id,emp_lname
             from employee", SQL_NTS );
SQLBindCol( stmt, 1, SQL_C_LONG, &emp_id,
             sizeof(emp_id), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
             sizeof(emp_lname), NULL );

for(;;) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
   print_employee( emp_id, emp_lname);
}

/* Using SQL_CLOSE closes the cursor
   but does not free the statement */
SQLFreeStmt( stmt, SQL_CLOSE );
. . .
```

Notes

- ODBC documentation suggests that you use SELECT... FOR UPDATE to indicate that a query is updateable using positioned operations. You do not need to use the FOR UPDATE clause in Adaptive Server Anywhere; SELECT statements are automatically updateable as long as the underlying query supports updates. That is to say, as long as a data modification statement on the columns in the result is meaningful, then positioned data modification statements can be carried out on the cursor.
- ODBC 3.0 provides a cursor type called a **block cursor**. When you use a block cursor, you can use **`SQLFetch`** to fetch a block of rows, rather than a single row.
- There are two alternatives for carrying out positioned updates and deletes in ODBC. You can send positioned UPDATE and DELETE statements using **`SQLExecute`** or you can use **`SQLSetPos`** to carry out the operation. Depending on the parameters supplied (SQL_POSITION, SQL_REFRESH, SQL_UPDATE, SQL_DELETE) **`SQLSetPos`** sets the cursor position and allows an application to refresh data, or update or delete data in the result set.

  With Adaptive Server Anywhere, you should use **`SQLSetPos`** to carry out positioned operations on cursors.

Bookmarks and cursors

ODBC provides **bookmarks**, which are values used to identify rows in a cursor. Adaptive Server Anywhere supports bookmarks for all kinds of cursor except dynamic cursors.

Before ODBC 3.0, a database could specify only whether it supports bookmarks or not. There was no way for a database server to indicate for what kind of cursor bookmarks were supported. Adaptive Server Anywhere returns that it does support bookmarks. There is therefore nothing in ODBC to prevent you from trying to use bookmarks with dynamic cursors; however, you should not use this combination.

Calling stored procedures in ODBC

This section describes how to call stored procedures and process the results from an ODBC application.

☞ For a full description of stored procedures and triggers, see "Using Procedures, Triggers, and Batches" on page 221 of the book *Adaptive Server Anywhere User's Guide*.

Procedures and result sets

There are two types of procedures: those that return result sets and those that do not. You can use **SQLNumResultCols** to tell the difference: the number of result columns is zero if the procedure does not return a result set. If there is a result set, you can fetch the values using **SQLFetch** or **SQLExtendedFetch** just like any other cursor.

Parameters to procedures should be passed using parameter markers (question marks). Use **SQLSetParam** to assign a storage area for each parameter marker, whether it is an INPUT, OUTPUT or INOUT parameter.

In order to handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the stored procedure definition. To avoid this problem, you can use column aliases in your procedure result set cursor.

An example with no result set

The following example creates and calls a procedure. The procedure takes one INOUT parameter, and increments its value. In the example, the variable **num_col** will have the value zero, since the procedure does not return a result set. Error checking has been omitted, to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
long i;
SWORD num_col;

/* Create a procedure */
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
       "CREATE PROCEDURE Increment( INOUT a INT )" \
       " BEGIN" \
          " SET a = a + 1" \
       " END", SQL_NTS );

/* Call the procedure to increment 'i' */
i = 1;
SQLSetParam( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0,
             0, &i, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )",
             SQL_NTS );
```

```
SQLNumResultCols( stmt, &num_col );
do_something( i );
```

An example with a result set

The following example calls a procedure that returns a result set. In the example, the variable **num_col** will have the value 2, since the procedure returns a result set with two columns. Again, error checking has been omitted, to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char emp_id[ 10 ];
char emp_lame[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
      "CREATE PROCEDURE employees()" \
      " RESULT( emp_id CHAR(10), emp_lname CHAR(20))"\
      " BEGIN" \
      " SELECT emp_id, emp_lame FROM employee" \
      " END", SQL_NTS );

/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &emp_id,
            sizeof(emp_id), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
            sizeof(emp_lname), NULL );

for( ;; ) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) {
   retcode = SQLMoreResults( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
} else {
   do_something( emp_id, emp_lname );
}
}
```

A sample program

A sample ODBC program, *odbc.c*, is supplied in the *cxmp* subdirectory of the Adaptive Server Anywhere installation directory. The program performs the same actions as the Embedded SQL dynamic cursor example program.

For a description of the associated Embedded SQL program, see "Database examples" on page 72.

Building the sample program

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 odbc {Platform} {Compiler}
```

The first parameter is **odbc**, meaning compile the ODBC example. The same batch file also compiles the Embedded SQL programs.

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

- **WINDOWS** compile for 16-bit Windows.
- **WIN32** compile for 32-bit Windows using the Watcom C 32-bit Windows support.
- **WINNT** compile for Windows NT or Windows 95.

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

- **WC** use Watcom C or Watcom C/32
- **MC** use Microsoft C
- **BC** use Borland C
- **CS** use IBM C Set++

Building the sample program as an NT service

The example program *odbc.c*, when compiled for Windows NT, runs optionally as a service.

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 the linked executable to be run either as a regular executable or as an NT service.

❖ **To run the compiled example as a Windows NT service:**

1. Start Sybase Central and open the Services folder.
2. Click Add Service. Follow the instructions for adding the sample program as a service.

3 Right-click the service icon and click Start to start the service.

When run as a service, the program displays the normal user interface if possible. It also writes the output to the Application Event Log. If it is not possible to start the user interface, the program prints one page of data to the Application Event Log and stops.

This example has been tested with the Watcom C/C++ 10.5 compiler and the Microsoft Visual C++ 2.0 compiler.

Using ODBC without the driver manager

ODBC applications do not generally link directly to the ODBC driver. Instead, they link to the **ODBC driver manager**, which in turn handles the driver or drivers that may be required.

On Windows operating systems, the driver manager is supplied by Microsoft. On other operating systems, other vendors do supply ODBC driver managers; for example, Intersolv and Visigenic supply ODBC driver managers for UNIX. However, driver managers are less generally available than on Windows operating systems.

It is also possible to develop applications that link directly to the ODBC driver. This is particularly useful on operating systems such as UNIX, where your end users may not have access to an ODBC driver manager.

You develop an application that uses the ODBC driver directly by carrying out the following steps:

- Use the supplied import library as usual.

  For descriptions of the import libraries, see "Compiling and linking an ODBC application" on page 128.

- Specify the DRIVER= parameter in the **`szConnStrIn`** argument to the **`SQLDriverConnect`** function. For example:

  ```
  szConnStrIn =
  "driver=ospath/dbodbc6.dll;dbf=c:\asademo.db"
  ```

  where *ospath* is the operating system subdirectory of your Adaptive Server Anywhere installation directory.

Developing ODBC applications on UNIX

An ODBC driver manager is not included with Adaptive Server Anywhere, but there are third party driver managers available. An ODBC driver manager includes the following files:

Operating system	Files
Solaris/Sparc, AIX	*libodbc.so* (*libodbc.so.1*)
	libodbcinst.so (*libodbcinst.so.1*)
HP-UX	*libodbc.sl* (*libodbc.sl.1*)
	libodbcinst.sl (*libodbcinst.sl.1*)

The Adaptive Server Anywhere ODBC driver is the following file:

Operating system	ODBC driver
Solaris/Sparc, AIX	*saodbc.so*
	saodbc_r.so (*saodbc_r.so.1*)
HP-UX	*saodbc.sl*
	saodbc_r.sl (*saodbc_r.sl.1*)

The libraries with the *_r* are for use with multi-threaded applications. If you are developing single-threaded applications, use the library without the _r.

If you want to use the Adaptive Server Anywhere ODBC driver without an ODBC driver manager you can do so, but you can then access only Adaptive Server Anywhere data sources. The libraries are installed as symbolic links to the shared library with a version number (in parentheses).

❖ **To use Adaptive Server Anywhere without an ODBC driver manager:**

- If you are creating a custom ODBC application, you can link directly to the Adaptive Server Anywhere ODBC driver.
- If your application requires an ODBC driver manager, create symbolic links for both the *libodbc* and *libodbcinst* shared libraries to the Adaptive Server Anywhere ODBC driver.

Data source information

If an ODBC driver manager is present, Adaptive Server Anywhere queries the driver manager for data source information. If Adaptive Server Anywhere does not detect the presence of an ODBC driver manager, it uses *~/.odbc.ini* for data source information.