**ODBC Driver Development**

**Open your database system to the world. Vikash steps through developing an ODBC driver, using XML-SOAP for encoding the calls and results. The communication between the driver and DBMS takes place over sockets.**

The primary design goal of ODBC (Open Database Connectivity) is to allow an application to access different databases, simultaneously, with the same source code.

An ODBC client always calls a fixed set of APIs to access data from any ODBC-compliant DBMS. The APIs for each DBMS together form an ODBC driver for that DBMS. The client does not statically link with any ODBC driver; in Windows, the driver is implemented as a DLL allowing it to be loaded at run time. The name, parameters, and what these APIs/functions are expected to do is defined by the ODBC standard. The use of drivers isolates applications from database-specific calls in the same way that printer drivers isolate word processing programs from printer-specific commands.

The aim of this article is to help you create an ODBC driver for your DBMS. I will discuss the ODBC architecture and then create a simple ODBC client. This will give you an idea of what is expected from your driver. Then we will proceed to create the driver itself.

I developed my ODBC driver on Windows 2000 professional using Visual C++ 6.0, so all the extra tools and samples mentioned are from Microsoft unless otherwise noted. Microsoft groups ODBC together with its other data access technologies under MDAC (Microsoft Data Access Components). MDAC is available as part of Platform SDK or as a separate download at<http://www.microsoft.com/data/whatcom.htm>.

#### ODBC Driver and Architecture

The ODBC architecture has four components:

* The *Application* performs processing and calls ODBC functions to submit SQL statements and retrieve results.
* The*Driver Manager* is a DLL provided on a Windows platform as part of the ODBC components. It manages certain tasks common to all ODBC clients. For example, it manages loading and unloading of driver DLLs, and creation and maintenance of pointers to driver functions so you don’t have to use**LoadLibrary** and **GetProcAddress** for each driver you want to use. It performs some basic error checking before a call is forwarded to the driver and also implements certain functions like**SQLDataSources**, **SQLDrivers**, and **SQLGetFunctions** within itself.
* The *Driver*processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application’s request so that the request conforms to syntax supported by the associated DBMS.
* The *Data source*consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.

#### Using a Data Source

A data source is simply the source of the data. It can be a file, a particular database on a DBMS, or even a live data feed. For example, a data source might be an Oracle DBMS running on an OS/2 operating system, accessed by Novell Netware; an IBM DB2 DBMS accessed through a gateway; a collection of Xbase files in a server directory; or a local Microsoft Access database file. The purpose of a data source is to gather all of the technical information needed to access the data — the driver name, network address, network software, and so on — into a single place and hide it from the user. The user should be able to look at a list that includes Payroll, Inventory, and Personnel, choose Payroll from the list, and have the application connect to the payroll data, all without knowing where the payroll data resides or how the application got to it. File data sources are stored in a file and allow connection information to be used repeatedly by a single user or shared among several users. When a file data source is used, the Driver Manager makes the connection to the data source using the information in a .dsn file. This file can be manipulated like any other file. A file data source does not have a data source name, as does a machine data source, and is not registered to any one user (UserDSN) or machine (SystemDSN).

A file data source streamlines the connection process, because the .dsn file contains the connection string that would otherwise have to be built for a call to the **SQLDriverConnect**function. Another advantage of the .dsn file is that it can be copied to any machine, so identical data sources can be used by many machines as long as they have the appropriate driver installed. A file data source can also be shared by applications. A shareable file data source can be placed on a network and used simultaneously by multiple applications. DSNs created for individual users will be called User DSNs are registered in the following system information key:

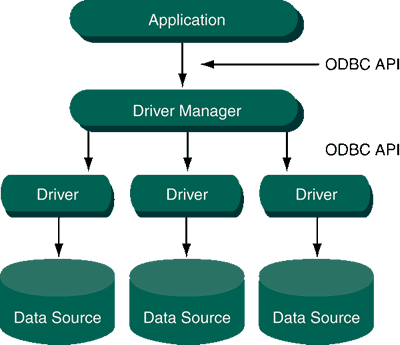
[?](http://drdobbs.com/windows/184416434)

|  |  |
| --- | --- |
| 1  2  3  4 | HKEY\_CURRENT\_USER      SOFTWARE          ODBC              Odbc.ini |

Similarly, system DSNs are tied up to a particular system, and any user of that system can use them. System DSNs are registered in the following system information key:

[?](http://drdobbs.com/windows/184416434)

|  |  |
| --- | --- |
| 1  2  3  4 | HKEY\_LOCAL\_MACHINE      SOFTWARE          ODBC              Odbc.ini |



**Figure 1: The ODBC architecture.**

The ODBC API is used in two places: between the application and the Driver Manager, and between the Driver Manager and each driver. The communication between the driver and the data source is the individual vendor’s choice. Note that you can create, implement, and use a driver without the driver manager, so do not worry about the typical “too many layers” situation.

The driver packages the API calls along with the parameters and sends it to the DBMS or data source. The DBMS processes the call and sends the results back to the driver, which throws back the results to the ODBC clients. I am going to use XML-SOAP for encoding the calls and results. The communication between the driver and DBMS takes place over sockets. We will discuss more on this in the section “Packaging calls in XML-SOAP and communicating over sockets”. You will need to have some idea about XML and sockets in addition to general C programming and DLL creation.

To make your database server work with your driver, a thin layer has to be added to it. This layer understands the XML-SOAP encoded calls from the driver; it interprets them to existing functions in your DBMS engine and encodes back the results as XML-SOAP. The core of this layer is an XML parser. The parser is also required as part of your driver. I have provided one in the download, but you are free to use one of your choice with necessary changes to the sample code.

It is important to understand that ODBC is designed to expose database capabilities, not to supplement them. It makes your DBMS accessible to more clients through a commonly accepted standard. One exception to this is a file-based driver, which works with raw data and therefore also implements the functionality. A driver working on a DBF file may or may not implement update and is not bound by any other engine. However, the driver for SQL Server is limited by the functionality provided by the SQL server engine, since the driver itself never touches the raw data files.

I am going to create a driver for a server based DBMS, since the aim is only to expose existing functionality. Besides, I feel that a file-based DBMS has little to offer in today’s client-server world.

Page2:

#### A Simple ODBC Client

In this section, I’ll cover the bare minimum ODBC calls required for fetching data from any ODBC data source. I’ll also discuss the three handle types or levels involved in communication between an application and an ODBC driver/driver manager. It’s important to be familiar with the ODBC administrator and to know the header files, libraries, and DLLs required for client and driver development.

**Example 1.**

[?](http://drdobbs.com/windows/184416434?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | // ----------- allocate ENVIRONMENT  status = SQLAllocHandle ( SQL\_HANDLE\_ENV, SQL\_NULL\_HANDLE, &hEnv );    // ----------- set the ODBC version for behavior expected------- this is compulsory  status = SQLSetEnvAttr ( hEnv,  SQL\_ATTR\_ODBC\_VERSION, (SQLPOINTER)SQL\_OV\_ODBC2, 0 );    // ----------- allocate CONNECTION  status = SQLAllocHandle ( SQL\_HANDLE\_DBC, hEnv, &hConn );    #ifdef  \_CONNECT\_WITH\_PROMPT    // ----------- actual connection takes place at this point  // ----------- option 1: user is prompted for DSN & options  status = SQLDriverConnect ( hConn, GetDesktopWindow(),                  ( unsigned char\* )"",                  SQL\_NTS, szConnStrOut, 1024, &x,                  SQL\_DRIVER\_COMPLETE );    #else         // ----------- OR    // ----------- actual connection takes place at this point  // ----------- option 2: all connection params for SQL server driver specified  status = SQLDriverConnect ( hConn, GetDesktopWindow(),                  ( unsigned char\* )"DRIVER={SQL Server};SERVER=MYSERVER;<br>                                    DATABASE=pubs;UID=sa;PWD=sa",                  SQL\_NTS, szConnStrOut, 1024, &x,                  SQL\_DRIVER\_COMPLETE );    #endif    // ----------- CONGRATUALTIONS ---- u r connected to a DBMS via an ODBC driver    // ----------- allocate STATEMENT  status = SQLAllocHandle ( SQL\_HANDLE\_STMT, hConn, &hStmt );    // ----------- execute the statement specified on command line  status = SQLExecDirect ( hStmt, ( unsigned char\* )argv[1], SQL\_NTS );    // ----------- RESULTS READY    // ----------- get the title of first col ( first 64 bytes only )  status = SQLColAttribute ( hStmt, 1, SQL\_DESC\_NAME, szColTitle, sizeof(szColTitle), &x, NULL );    // ----------- bind buffer for obtaining column data ( first 255 bytes only )  status = SQLBindCol ( hStmt, 1, SQL\_C\_TCHAR, (SQLPOINTER)szColData, <br>                      sizeof(szColData), &i );    // ----------- fetch a row i.e. the first row  status = SQLFetch ( hStmt );    // ----------- show the title and value of the first col, first row  printf ( "Row:1, Col:1    Title: %s,   Value: %s", szColTitle, szColData ); |

Example 1 shows the minimum nine calls required to connect, execute a query, specify a buffer for results, and get the title and value of the first column of the first row. Error checking has been omitted for brevity. We discuss error handling as expected from an ODBC driver in the diagnostics section. A more complete client is available as download — CLIENT.CPP. Although the client you get with the download shows you all the rows and columns obtained on executing the query, it still uses these nine calls only, so you can start with these, but clients like MS Query and MS SQL Server DTS expect more from your driver.

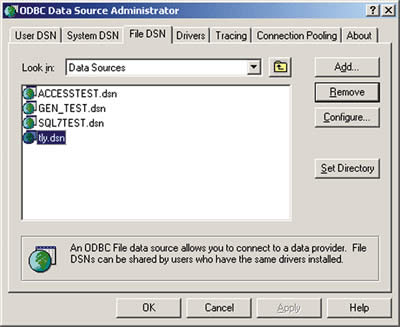
The ODBC driver provides **SQLAllocHandle** and **SQLFreeHandle** to allocate and free all three types of handles — environment, connection, and statement. The three handles can be described as follows:

* The **environment handle** encapsulates one or more connections along with global information like version for expected ODBC behavior from the driver and also the environment level diagnostics.
* The **connection handle** represents a connection to the DBMS/data source along with connection specific information like connection timeout, transaction isolation, etc. and the connection level diagnostics.
* The **statement handle** can be thought of as a single query like **SELECT \* FROM employee**. As we can see from Example 1, this is the handle used to execute the query **SQLExecDirect**, get information about the result **SQLColAttribute**, specify buffers in which the driver can feed the results **SQLBindCol**, and finally fetch the rows **SQLFetch**. All three are internally represented as C structures in our sample driver GODBC.H.
* **SQLDriverConnect** is the API that actually helps you choose a driver, specify the connection parameters and make a connection. Before making a connection, two things are required — a driver and the connection parameters for that driver to connect to the DBMS or data source. I emphasize this because as you work with ODBC, you will find a number of ways in which these are collected. The Driver Manager, Driver, DSN, ODBC Administrator, and user prompts all combine to achieve this simple goal. From the perspective of a driver writer, you will get the connection parameters for your DBMS or you will need to prompt the user for it.

There are other connection APIs like **SQLConnect** and**SQLBrowseConnect** but **SQLDriverConnect** has the following advantages over **SQLConnect**:

* To let the application use driver-specific connection information
* To request that the driver prompt the user for connection information
* To connect without specifying a data source

In Example 1, you can see that I have given you two possible ways the **SQLDriverConnect** API can be called. In the first case, we specify an empty string as connection parameters. The driver manager helps us choose a driver in the form of a DSN as shown in Figure 2.



**Figure 2: DSN selection using ODBC administrator.**

The DSN is a collection of settings for making an ODBC connection. The most practical being the File DSN, a simple text file which allows you to specify the driver and parameters in the form of key-value pairs. For example:

[?](http://drdobbs.com/windows/184416434?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4 | DRIVER=SQL Server  SERVER=MYSERVER  DATABASE=  UID=sa |

These are concatenated by Driver Manager and sent to your driver. The second option is to specify the driver and connection parameters yourself, as I have done for my SQL server **"DRIVER={SQL Server};SERVER=MYSERVER;DATABASE=pubs;UID=sa;PWD=sa"**. You need to parse these key-value pairs in your driver, and if anything is missing or something is required you need to prompt the user for it; password being the most relevant example. The download has a file GO\_CONN.CPP dedicated to this API.

Figure 2 shows the ODBC administrator, as invoked by the Driver Manager when you call **SQLDriverConnect** with empty strings as in-connection parameters. You will need to walk through the tabs to get an idea about ODBC on your system. The different types of DSNs are basically different ways to store or scope the connection settings, for example restrict it to a user (UserDSN) or machine (SystemDSN). The File DSN is portable and the most usable. The drivers tab as we will see is going to show our driver the **General ODBC driver 0**along with other drivers installed on the system. The tracing option is discussed in the troubleshooting section. The About tab is important, don’t skip that, it tells you the files, which constitute your driver manager, ODBC administrator, and other core components of ODBC on the MS platform. As you will see, odbc32.dll and odbccp32.dll are the most relevant ones for us now.

The client includes the SQL.H and SQLEXT.H header files. SQL.H contains the ODBC function prototypes along with definition for constants used to get basic information about the driver at the three levels. This file in turn includes the SQLTYPES.H, which as the name indicates contains definitions for data types used in ODBC like SQLHANDLE, SQLRETURN, etc. SQLEXT.H defines more constants required for working with ODBC and is an extension to SQL.H as the name indicates. Also required is the ODBC32.LIB. This is the import library for the driver manager. You are going to use the same header files when you develop your driver. Note that the driver does not require these LIBs or DLLs to be compiled, but a client typically uses the driver in conjunction with the Driver Manager (odbc32.dll) and ODBC administrator (odbccp32.dll).

#### Your First Driver and Its Recognition

I divide the driver development into two stages. First, develop a skeleton driver. This is to know the basic structure, compilation process, installation process, and registry settings and also to test communication between client, driver manager, and ODBC administrator. Then I discuss how the functions are implemented to make it a real-life driver. The download has folders ODBCDRV0 and ODBCDRV1 representing this division.

I will use two files to compile my first ODBC driver, ODBCDRV0.C and ODBCDRV0.DEF, and then register this compiled driver DLL (ODBCDRV0.DLL) as **General ODBC Driver**. Finally, I create a FileDSN and test connectivity with ODBC Administrator. ODBCDRV0.C contains all the ODBC functions with bare-minimum body with the exception of **DllMain** and **SQLDriverConnect**(shown in Example 2).

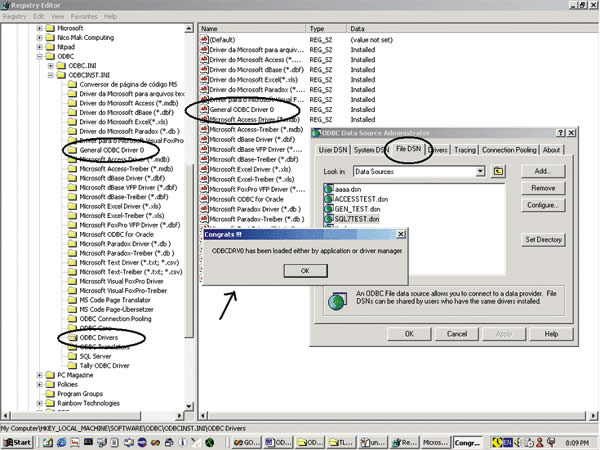
<="" b="">

[?](http://drdobbs.com/windows/184416434?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | //////  DLL entry point for global initializations/finalizations if any    BOOL WINAPI DllMain ( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved )  {      if ( fdwReason == DLL\_PROCESS\_ATTACH ) {        // DLL is being loaded            MessageBox ( GetDesktopWindow(),              "ODBCDRV0 loaded by application or driver manager",              "Congrats !!! ", MB\_OK);            // any initialization global to DLL goes here        }      else if ( fdwReason == DLL\_PROCESS\_DETACH ) {   // DLL is being unloaded            // any finalization global to DLL goes here            MessageBox ( GetDesktopWindow(), "ODBCDRV0 is being unloaded",              "Congrats !!! ", MB\_OK);      }      return TRUE;  }    ///// SQLDriverConnect    RETCODE SQL\_API SQLDriverConnect (      HDBC             hDBC,                       HWND     hWnd,      UCHAR\*         szConnStrIn,            SWORD   cbConnStrIn,      UCHAR\*         szConnStrOut,         SWORD   cbConnStrOut,      SWORD\*        pcbConnStrOut,      UWORD    uwMode )  {      OutputDebugString ( "SQLDriverConnect called\n" ); // for DBMON        if ( cbConnStrIn == SQL\_NTS && szConnStrIn ) // get in-string length          cbConnStrIn = strlen ( szConnStrIn );        MessageBox ( hWnd, "Connection dialog would go here",                                          "Sample driver",  MB\_OK );        // copy in-conn string to out string      if( szConnStrOut && cbConnStrOut > 0 ) {            strncpy ( szConnStrOut, szConnStrIn,                           ( cbConnStrIn == SQL\_NTS ) ? cbConnStrOut - 1 :                                               min( cbConnStrOut, cbConnStrIn ));            szConnStrOut[cbConnStrOut - 1] = '\0';      }        if ( pcbConnStrOut )      // set length of out string also          pcbConnStrOut = cbConnStrIn;        return SQL\_SUCCESS;  }      ///// SQLExecDirect /////    RETCODE SQL\_API SQLExecDirect (              HSTMT                   hStmt,              UCHAR\*                 sStmtText,              SDWORD               iStmtLen )  {      OutputDebugString ( "SQLExecDirect called\n" );      return SQL\_SUCCESS;  }    ///// SQLAllocHandle /////    RETCODE SQL\_API SQLAllocHandle  (               SQLSMALLINT     HandleType,               SQLHANDLE        HandleParent,               SQLHANDLE\*       NewHandlePointer  )  {      OutputDebugString ( "SQLAllocHandle called\n" );      return SQL\_SUCCESS;  } |

All the other functions contain a call to **OutputDebugString** that can be tracked through **DBMON.EXE**. You can use **MessageBox** if you like instead of **OutputDebugString**. These are dummy implementations just like **SQLExecDirect** and **SQLAllocHandle**shown in Example 2.

**DllMain** shows a message when the application or driver manager loads or unloads our driver DLL (shown in Figure 3).



**Figure 3: Driver registration and first connection.**

**SQLDriverConnect** copies the in-connection string to out-connection string with due respect to their lengths so that the caller does not reject the driver as invalid. The DEF file is a standard way of exporting functions from a DLL and is a simple text file used by the linker. Example 3 shows how a DEF file looks.

**Example 3.**

[?](http://drdobbs.com/windows/184416434?pgno=2)

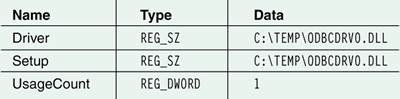
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | LIBRARY   ODBCDRV0  DESCRIPTION "General ODBC Driver 0"  EXPORTS              SQLAllocConnect              SQLAllocEnv              SQLAllocStmt              .              .              .              all function to be exported |

Note that you are not restricted to using any files I have previously mentioned. Their purpose is to make things easy for you. You can write both the files on your own and use Example 2 if you like. The problem is picking up each function prototype from SQL.H file and implementing it by putting something like **MessageBox** or**OutputDebugString** into it. Writing DEF is a trivial job; you merely need to list the function names and put a few compulsory lines like the LIBRARY statement on the top.

After you compile the DLL, the next step is to register it so that the system and clients recognize it. I suggest you copy the DLL to the \TEMP folder of your drive to keep the registered driver separate from the one being updated and compiled. You can register the driver using either the API call **SQLInstallDriverEx** or you can manually change the registry; its not very difficult. I will explain both.

Start REGEDIT.EXE. Navigate to HKEY\_LOCAL\_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers and create a new String Value with Name **General** **ODBCDriver** **0** and Data **Installed**. Figure 3 shows REGEDIT working. This adds your driver to the list of installed drivers on your system. Now navigate up one level to HKEY\_LOCAL\_MACHINE\SOFTWARE\ODBC\ODBCINST.INI and create a new key under it. Name it **General** **ODBC** **Driver** **0**. Under this key create the three Name-Data pairs shown in Table 1, which basically tell the system the name and location of your driver DLL.

**Table 1.**



**Table 1: Three Name-Data pairs describing the name and location of your driver.**

You are now ready to test a connection to your driver. The system has the following info about our driver:

* It knows where to find the DLL for the driver
* It knows that the same DLL also implements configuration functions and therefore acts as the Setup DLL for the driver
* The number of times the driver has been installed is one.

Defining some more values in the registry like **ConnectFunction**and **APILevel** can further refine the configuration, but it’s your choice. In most cases, the default should be good enough. Let us now have a look at **SQLInstallDriverEx**.

**SQLInstallDriverEx** adds information about the driver to the Odbcinst.ini entry in the system information and increments the driver’s **UsageCount** by one. This function does not actually copy any files. You can do it yourself or through a setup program. The first two parameters to the function are important. The first parameter is made up of key value pairs separated by a zero char and the last pair is terminated with an extra zero char. The format can be defined as follows:

[?](http://drdobbs.com/windows/184416434?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5 | driver-desc\0      Driver=driver-DLL-filename\0      [Setup=setup-DLL-filename\0]      [driver-attr-keyword1=value1\0]      [driver-attr-keyword2=value2\0]...\0 |

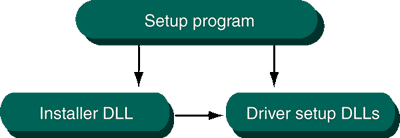
and example for our driver is:

[?](http://drdobbs.com/windows/184416434?pgno=2)

|  |  |
| --- | --- |
| 1  2  3 | General ODBC Driver\0      Driver=ODBCDRV0.DLL\0      Setup=ODBCDRV0.DLL\0\0 |

The second parameter is the path to our driver DLL, which is C:\TEMP, where the drive letter is changed as per your system. I have provided an ODBCREG.CPP file as an example for using this API. The corresponding API for removal is **SQL RemoveDriver**. The LIB required for this API is ODBCCP32.LIB and the corresponding header file is ODBCINST.H.

ODBC files are basically divided into two categories which is also how your driver should be arranged. One set of files, like ODBCCP32.LIB and ODBCINST.H, are for installation and other administration issues. The other, like the Driver Manager ODBC32.LIB and SQL.H, participate in the core functionality. I mention this since I have created a String-Value called SETUP during our installation using REGEDIT. The two functions**ConfigDriver** and **ConfigDSN** make the Setup DLL, which can be used to configure your driver and associated DSNs. You can implement it separately or within the same DLL. The setup DLL and its functions are accessed via ODBCCP32.DLL just like your core driver functions are accessed via ODBC32.DLL. I show the relationship in Figure 4. To start, we can rely on the native Installer DLL — ODBCCP32.LIB for all our needs.



**Figure 4: Installer DLL and driver Setup DLL relationship.**

To see if everything worked properly, start the ODBC Administrator (odbcad32.exe) from the Control Panel and go to the Drivers Tab. You will find the **General** **ODBC** **Driver** **0** listed there. Now proceed to make the first connection. Go to the FileDSN TAB and choose Add. You will get a list of drivers to choose from; choose our Driver and click Next. Specify any name for this FileDSN you are creating; this will be the name of the file in which the settings will be stored. GDSN0 is a good choice. You are notified that a FileDSN is about to be created for your driver. Click finish. You should get the message box, which is written in **DllMain** with the message **ODBCDRV0loaded** **by** **application/Driver** **Manager**. Congratulations! Immediately following this the ODBC Administrator calls the**SQLDriverConnect** API which should give you another message box telling you **Connection** **dialog** **would** **go** **here**. We have covered our first major milestone. Now let us see how the function calls are encoded and sent to the database server and response from the server sent back to the client.

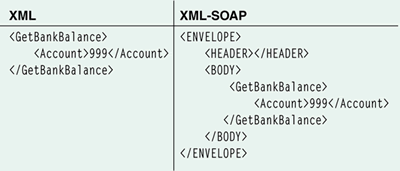
Page 3:

#### Packaging Calls in XML-SOAP and Communicating Over Sockets

I mentioned earlier that the calls received by your driver are packaged using XML-SOAP and communicated to the server via sockets. The server responds with an XML-SOAP response on the same socket, which is interpreted by the driver and then sent to the application as per the ODBC standard. The key point is that the communication style between your driver and datasource or database server is your choice. The application is not interested in knowing how you obtain the data; it just expects it in the buffers it has specified and in a manner defined by ODBC.

XML is a platform-neutral data representation protocol. I believe that XML has two attributes that make it unique. First, it is text based and second, it allows hierarchical data to be represented easily. By using XML, data can be serialized into a transmissible form that is easily decoded on any platform. SOAP provides an encoding scheme over XML and thus inherits all the advantages of XML. There is no existence of SOAP separate from XML. The prime target of SOAP is interoperatibility while using XML encoding for request-response in IPC/RPC. In the SOAP framework, few XML tags are required, and data is serialized in a standard fashion as per the type of data. This implies that sender and receiver do not need to negotiate the common parts of every communication.

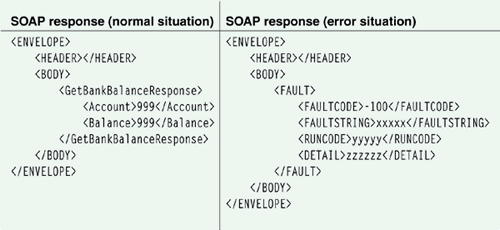
Table 2 shows a call to function **GetAccountBalance** with one parameter i.e. account number encoded in XML as well as XML-SOAP.



**Table 2: Encoding a call in XML and SOAP/XML.**

I have kept the tags required by SOAP in uppercase. As you can see from Table 2, SOAP requires you to put your request-response in an **ENVELOPE** tag and divide it into two parts with **HEADER** and**BODY** tags.

When the response is constructed, the function or method name is appended with a string — **RESPONSE**, so **GetBankBalance**becomes **GetBankBalanceResponse**. If an error situation arises, a**FAULT** tag is used instead of the method name with details of the error like code, message etc. Table 3 shows a response on success and a response on error.



**Table 3: SOAP response for success and error.**

That is almost all about SOAP. You may be surprised, but this relatively new and simple looking text-based protocol is challenging COM and CORBA as far as RPCs are concerned. The clear advantage being ease of design and implementation, compatibility across various systems, ability to piggyback on protocols like HTTP, and to perform RPC even on the Internet.

One thing I have left out purposely is the way complex or compound data types like arrays or structures are encoded in SOAP. If you are interested in learning more on this, you can check the latest [SOAP specification](http://www.w3.org/TR/2000/NOTE-SOAP-20000508/) on w3c site or read the excellent article by Don Box —[A Young Person’s Guide to the Simple Object Access Protocol](http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp). Now that we have some idea about encoded calls and responses, let’s see how sockets transport these across process, machine, and network boundaries.

A socket represents an endpoint for communication between processes across a network transport. The two endpoints are connected through a communication link. The communication link is an abstract expression for everything involved in transmitting data. Understanding sockets becomes very easy if you think you are opening a file, which you can write to or read from. The difference is that the name of the file in this case is a network address and a port number, so instead of **test.txt** you have something like**127.0.0.1:80**. Also, the server serves the content of this file hot. The socket API has been there and has almost remained unchanged since the early Unix days (hence the common name Berkeley sockets).

You do not have to worry about how sockets are implemented. As long as TCP/IP is installed and you are able to see a machine on the LAN or access it on the Internet, you should be able to open a connection to it using a socket API subject to the condition, that there should be a listener on the other end. To make things clear, let’s take an everyday example of sockets in action. Internet Explorer, or any of your browsers, opens a socket connection to the site using its IP address and port 80, which is the standard port for HTTP communication. What follows is simple socket IO. The steps in the process are:

1. Browser opens a socket connection with server address and port 80
2. Browser writes the request for a resource (e.g., an HTML page) to the socket
3. Web-server reads the request from the socket and prepares a response (e.g., the requested HTML page)
4. Web-server writes back the response to the socket
5. Browser reads it, interprets it as per Content-Type, and renders it as necessary

The browser here is a socket client and the web-server is the socket server. Our ODBC driver acts as a socket client and communicates with our database server, which acts as a socket server.

Let’s take an example in context of our driver on how this whole process of request response works. The client calls **SQLExecDirect**for the query **SELECT \* FROM** **authors**. I will show you the actual encoding which the driver ODBCDRV1 (provided with the download) is going to do for this call and also the response template, since the actual data in response is dependent on the server.

By looking at Example 4, the use of **ENVELOPE**, **HEADER**, and**BODY** tags must be obvious as part of a SOAP request. I have put a connection ID in the header. Since the server would need to know which client is making the request, the client should always send this information with the request. The header seemed a good place to me, but again SOAP does not specify anything, it’s your choice. Note that the samples I have provided as download do not use this ID and work in a stateless fashion to keep things simple.

**Example 4.**

[?](http://drdobbs.com/windows/184416434?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <ENVELOPE>      <HEADER>          <CONNECTID>12345</CONNECTID>      </HEADER>      <BODY>          <SQLExecStmt TYPE="General">              <STMT>SELECT \* FROM authors</STMT>          </SQLExecStmt>      </BODY>  </ENVELOPE> |

The choice of method name is also between you and your server, so any name, which you program your server to recognize, will do. I have used **SQLExecStmt** but it could be any valid XML tag name or even **SQLExecDirect**, the name of the API. Parameters to functions are encoded as child elements of the method name element. The query statement is sent as STMT element, a child of the method name element. I have used the terms **tag** and **element**interchangeably throughout the text, but they both refer to any valid XML name inside angle brackets.

This is how most of the calls are encoded. Although you will see a reasonably long list of exported function calls in your DEF file, the types of call you need to encode are not many. You will see that a generic encoder with a few parameters will work well. Before we see how this request is sent to the server using a socket, let us go a little deeper and see how encoding is done. This is accomplished using the XML parser and the XML DOM (Document Object Model). A request, as shown in Example 4, is directly prepared using DOM programming while a response received as a stream of bytes is processed into DOM using the parser. I describe both of these cases below. A complete working example is available in the download as XMLTEST.

#### Encoding XML Request or Response Using DOM Methods

The Document Object Model is a platform and language-neutral interface that defines a document as a structure that can be programmatically manipulated with ease. If you want to search for the STMT tag and extract its immediate content, the DOM would expose a standard walk and search method for this. More information on this is available at **http://www.w3.org/DOM/**.

Since XML is hierarchical and also supports a list of attributes within an element, we have to provide a structure which can store a tree of elements as well as contain a linked list of attributes. I have implemented this as a C++ class **XMLNode** in the file XMLTREE.CPP. The request shown in Example 4 can be encoded with **ENVELOPE** as the root node containing two child elements**HEADER** and **BODY**. The **HEADER** in turn contains the**CONNECTID** and so on. Example 5 shows how this can be done.

**Example 5**

[?](http://drdobbs.com/windows/184416434?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | XMLNode\*        root;     XMLNode\*        node1;     XMLNode\*        node2;     XMLNode\*        node3;     XMLNode\*        nodeattr;       // create the root element     root = XMLNode::CreateElement ( "ENVELOPE", NULL );       // create the HEADER and append it to ENVELOPE     node1 = XMLNode::CreateElement ( "HEADER", NULL );     root->AppendChildNodeBeforeX( node1, NULL );       // create the CONNECTID and append it to HEADER     node2 = XMLNode::CreateElement ( "CONNECTID", "12345" );     node1->AppendChildNodeBeforeX( node2, NULL );       // create the BODY and append it to ENVELOPE     node1 = XMLNode::CreateElement ( "BODY", NULL );     root->AppendChildNodeBeforeX( node1, NULL );       // create methodname tag and append it to BODY     node2 = XMLNode::CreateElement ( "SQLExecStmt", NULL );     node1->AppendChildNodeBeforeX( node2, NULL );       // create STMT tag and append it to method name tag     node3 = XMLNode::CreateElement ( "STMT", "SELECT \* FROM authors" );     node2->AppendChildNodeBeforeX( node3, NULL );       // create and append the statement type as attribute     nodeattr = XMLNode::CreateAttribute ( "Type", "General" );     node1->AppendChildNodeBeforeX( nodeattr, NULL );       // stream the tree to a file     root->StreamToFile ( 0, \_FILE\_STDOUT, stdout ); |

The samples in the download are built around my parser and DOM implementations, but you are free to adopt anything of your choice like MS XML (Microsoft), Xerces (Apache Project), or Expat (James Clark) provided you are willing to change the remaining code. I suggest that you first be comfortable with ODBC driver development and then move on to changing these underlying tools and technologies.

The **root** node now encapsulates the complete request. The**CreateElement** is a static member function of the **XMLNode**class, which allocates an **XMLNode** object and assigns it the name specified in the first parameter. If the element is supposed to contain text, the user can specify the text as the second parameter. A child node is then allocated and its value set to this second parameter. The **AppendChildNodeBeforeX** is used to create associations between the nodes and allows you to append a specified **XMLNode** object as a child to another **XMLNode** object before the specified object from the already existing child nodes or as the last child if NULL is specified in the second parameter. The**StreamToFile** method can be used to stream the request to screen, file, or socket. This method walks through the tree and writes the content to the specified target. I have explained sockets conceptually earlier in this section; let’s see an example on how a socket client can be implemented.

#### Socket Client and Server

In Windows OS, you need to initialize the Winsock library before any other socket related calls are used. This call **WSAStartup** also requires you to specify the version (1.0,1.1,2.0) required by your program to work. Next, you create a socket using the API call**socket** specifying the address family (AF\_INET for Internet style addresses), type of socket (stream or datagram), and also the protocol to use (zero for using the available transport like TCP/IP). The socket is now created but is not connected. Specify the target server address along with the port and address family using the**SOCKADDR\_IN** structure defined in winsock.h in call to **connect**. This call actually connects you to the specified socket server.

You must have the correct IP address of the server and the port on which the server is listening or else the connection will fail. Also, you need to convert the port number to network byte order from the host storage so that it is independent of the platform on which the client or server is running. I mention this because Intel-based systems store the integer values in format different from what is used by the RISC-based systems. Also, if you have a DNS name like**www.yahoo.com**, you can use the **gethostbyname** API to convert it to an IP address. This API has not been shown in the example but is part of samples in the download.

Once connected, you can send the request using **send** and receive the response sent by the server using **recv**. After the request response session is complete, you can indicate the end of communication using **shutdown** and finally destroy the socket using **closesocket**. Example 6 shows this. It connects to the socket server on the local machine (127.0.0.1) on port 9999. I will explain below the implementation of a basic socket server to serve this client. The client will send a string “Hello server” to the server and the server sends back the string “Hello Client”. Error checking is not shown for brevity. You must link this program with mswsock.lib or ws2\_32.lib for it to work.

**Example 6.**

[?](http://drdobbs.com/windows/184416434?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | int                               status;  // status or return values  SOCKET                     sckt;           // socket handle like file handle  WSADATA                 wsadata;           // winsock requirement for startup  SOCKADDR\_IN         address;               // structure for IP, port etc.  char                           buf[128]   // buffer to recv response        // winsock initialization, specify the version required      status = WSAStartup ( MAKEWORD( 1, 1 ), &wsadata );        // create a socket, no address associated as yet      sckt = socket ( AF\_INET, SOCK\_STREAM, 0 );        // address details of server to communicate with      address.sin\_addr.s\_addr = inet\_addr ( "127.0.0.1" );      address.sin\_family            =  AF\_INET;            // address family      address.sin\_port               =  htons ( 9999 );    // port number        // connect to server analogous to opening a file      status = connect ( sckt, ( struct sockaddr\* )&address, sizeof(address));        // send a hello to server/listener      status = send ( sckt, "Hello server", strlen ( "Hello server" ), 0 );        // recv response from server (limited to 128 bytes )      status = recv ( sckt, buf, 128, 0 );        // initiate a shutdown for both send and recv      status = shutdown ( sckt, 0x02 );        // close the socket handle      status = closesocket( sckt );        // winsock finalization      WSACleanup (); |

To play around more, you can replace the localhost address 127.0.0.1 with the IP address of your favorite site, replace the port 9999 with standard HTTP port 80, and specify **GET**/**pagename.htm**\**r**\**n**\**r**\**n** in the send string. You will now be able to get at least the first 128 bytes (capacity of the variable in code snippet) of the specified HTML page typically seen using the browser.

This is how the **XMLNode** class used by the sample driver**ODBCDRV1** streams the XML request to the socket using the**TSocketClient** class. You will see that implementing a basic socket server is equally simple. Example 7 shows how we can create a socket server, which can serve our simple socket client, described above. The difference here is that instead of using the **connect**API after creating a socket, we bind it to an address and port available on the machine using **bind**. The address and port are specified through the SOCKADDR\_IN structure. Next, we set the socket in listening mode using the **listen** API. Note that the socket is now ready to listen and accept connections but is not yet accepting connections. The **accept** API is the call on which your program stops and waits for a connection. This API on the server side serves the **connect** API from the client. Another distinguishing factor is that when the client connects, the **accept** API creates and returns a new socket on which the communication with the client takes place. The first socket can continue listening for other connections; this is how a single server serves multiple clients. (The example does not show this.) If you already have a database server listening and serving sockets then you need not worry about this part.

**Example 7.**

[?](http://drdobbs.com/windows/184416434?pgno=3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | Int                          status;      // status or return values  Char                      buf[128];       // buffer to recv request  SOCKET               listen\_sckt;         // socket handle to listen  SOCKET               conn\_sckt;           // socket handle for IO  WSADATA            wsadata;               // winsock requirement for startup  SOCKADDR\_IN    listen\_addr;                // structure for IP, port etc.        // winsock initialization, specify the version required      status = WSAStartup ( MAKEWORD( 1, 1 ), &wsadata );        // create a socket, no address associated as yet      listen\_sckt = socket ( AF\_INET, SOCK\_STREAM, 0 );        // prepare to bind socket to a port on any IP for the machine      listen\_addr.sin\_family              = AF\_INET;          // address family      listen\_addr.sin\_port                  = htons ( 9999 ); // port number      listen\_addr.sin\_addr.s\_addr    = htonl ( 0 );           // any IP address        // bind the socket to the address family, port, IP      status = bind ( listen\_sckt, ( const struct sockaddr\* )&listen\_addr,                                sizeof(listen\_addr));        // switch to listen mode to allow connections      status = listen ( listen\_sckt, 1 );        // wait and accept connections      conn\_sckt = accept ( listen\_sckt, NULL, NULL );        // recv request from client      status = recv ( conn\_sckt, buf, 128, 0 );        // send a hello to client      status = send ( conn\_sckt, "Hello Client", 12, 0 );        // initiate a shutdown for both send and recv      status = shutdown ( conn\_sckt, 0x02 );        // close the client connection socket      status = closesocket( conn\_sckt );        // close the listening connection socket      status = closesocket( listen\_sckt ); |

I have described socket communication in the simplest form. There are two more important issues — synchronization and scalability. Synchronization defines how the client or server determines that the other has completed writing his request or response so that it can move ahead and act on it. There are two ways to solve this. One is the HTTP way of providing headers containing the length of content to follow. The actual content starts after the headers and an empty line. The other way is to have a fixed signature at the start and end of the content, something similar to MIME. I prefer the second way since our requests and responses are prepared in DOM and streamed directly while walking the tree. Determining the length of the content in advance would require walking the DOM tree twice, which would be cumbersome. I use a fixed signature of**\_\_\_\x4\x4MSG\_SOCK\x4\x4\_\_\_** defined as **MSG\_SOCK\_SIGN**in **SOCK\_CLI.HPP** and **SOCK\_SVR.HPP**. You can change it if you wish. A discussion on socket scalability is outside the purview of this article. Winsock provides a number of models depending upon the nature of application and scalability requirements. I suggest reading *Network Programming for Microsoft Windows*,by Anthony Jones and Jim Ohlund.

#### Parsing Streamed XML to DOM

The server receives the request as a stream of bytes from the driver, and the driver also receives the response from the server as a stream of bytes. This time, the DOM methods cannot be used to interpret this stream of bytes. The stream is sent through a parser, which checks and converts it to our DOM structure. After it is converted to DOM, it can then be accessed and manipulated using the DOM methods. Let’s take the response for our request shown in Example 4. I explain the response and the role of parser in interpreting it. The response is shown in Example 8.

**Example 8.**

[?](http://drdobbs.com/windows/184416434?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | <ENVELOPE>      <HEADER>          <CONNECTID>12345</CONNECTID>      </HEADER>      <BODY>          <SQLExecStmtResponse ResultType="Resultset">                <RESULTDESC>                  <ROWDESC>                      <COL>                          <NAME>author\_name<NAME>                          <ALIAS><ALIAS>                          <TYPE>varchar<TYPE>                          <LENGTH>64<LENGTH>                          <PRECISION><PRECISION>                          <NULLABILITY></NULLABILITY>                      </COL>                      <COL>                          <NAME>id<NAME>                          .                          .                      </COL>                  </ROWDESC>              </RESULTDESC>                <RESULTDATA>                  <ROW>                      <COL>Author1</COL>                      <COL>991</COL>                  </ROW>                  <ROW>                      <COL>Author2</COL>                      <COL>992</COL>                  </ROW>              <RESULTDATA>            </SQLExecStmtResponse>     </BODY>  </ENVELOPE> |

The typical response from the database server will be a resultset. I have divided it into two parts. The first part, Metadata, describes the resultset. This is contained inside the **RESULTDESC** tag. The second part is the data itself and is contained inside **RESULTDATA**tag.

Since I am describing a row of data in a resultset, I have used a**ROWDESC** tag inside the **RESULTDESC** tag. Each column is described with **COL** tag, which contains the six basic properties**NAME**, **ALIAS**, **TYPE**, **LENGTH**, **PRECISION**, and **NULLABILITY** for each column as child tags. Use of **ROWDESC** can be avoided and is a personal design choice; you can directly put the columns inside the **RESULTDESC** tag.

The **RESULTDATA** tag contains the data as rows and columns. Each row is contained in the **ROW** tag with the columns and values as child tags. The **RESULTDATA** contains as many **ROW** tags as there are rows in the resultset while each **ROW** contains as many**COL** tags as there are columns per row.

Example 8 shows how this response is converted into DOM before we serve it to our ODBC client. The parser is implemented as the**XMLParser** class and is available with the download. I will not go into the details of how the parser does its job. I have created it using YACC (Yet Another Compiler Compiler) and it goes through the stages typical of a parser — lexical, syntactic, semantic, analysis, and finally reduction of tags into my DOM class **XMLNode**. You can choose from one of the many parsers available in the market or create your own if you find this area of development interesting. We create an instance of the parser class with the buffer received on the socket connection and the size of the buffer. The **GetObjectNode** method parses it and returns an**XMLNode** object, which we can now use to interpret and serve the data.

Example 9 shows how you can serve the name of the second column (metadata) or data from the third column in the second row.

**Example 9.**

[?](http://drdobbs.com/windows/184416434?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | // allocate the XML parser with buffer to be parsed      xp = new XMLParser ( xbuf, size );        // parse and convert to DOM node      n = xp->GetObjNode ();        // release the parser      delete xp;        // check if parsing was successful      if ( n == NULL ) printf ( "failed\n" );        // check if the response is valid SOAP response      // code to check for ENVELOPE/HEADER/BODY etc        // store the full response form the server      pStmt->RespFull             = n;        // store the result descriptor ENVELOPE-BODY-RESULTDESC      pStmt->RespRSDesc   =  n->GetLastChild()->GetFirstChild();        // store the result data ENVELOPE-BODY-RESULTDATA          pStmt->RespRSData   = n->GetLastChild()->GetLastChild();        // initialize the state and position variables for fetch calls      pStmt->CurRowsetStartRow          = NULL;      // start of current rowset      pStmt->CurRowsetStartRowPos   = 0;             // absolute position      pStmt->CurRowsetEndRow           = NULL;       // end of current rowset      pStmt->CurRowsetEndRowPos     = 0;             // absolute position        // finally at some point in time      delete n; |

It is basically understanding of the tree structure and walking through the children and siblings. The column information is stored inside **ROWDESC**, which itself is the child of **RESULDESC**. We get the node for the first column using the **GetFirstChild** method on the **ROWDESC** node, which is itself obtained using the**GetFirstChild** method on **RESULTDESC** object. Now we loop using**GetNextSibling** method of the DOM since the columns are all immediate children of **ROWDESC** and are therefore siblings to one another. Once we reach the desired column, we now loop again through its children comparing the names of the tags with the desired property; i.e., **NAME**. We take the text stored in it and serve it to the client. Note here that the text content of an element is not the value obtained via the **GetNodeValue** method, instead it is the node value of the first child element. This is the design of the **XMLNode** class which keeps it flexible enough to store mixed content; i.e., both text and elements within a node. The client typically specifies the buffer in which it requires the value. The value is copied to this client buffer, see Example 10.

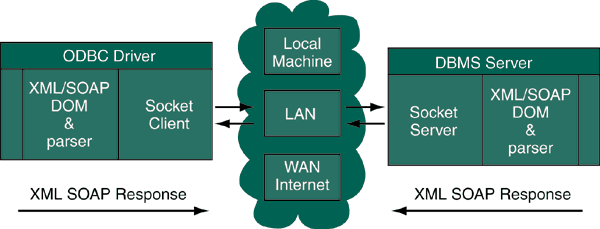
**Example 10.**

[?](http://drdobbs.com/windows/184416434?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | // HOW TO serve the name of the second col (metadata)    // 1. start with the first COL in ROWDESC  rowdesc =pStmt->RespRSDesc->GetFirstChild();  coldesc = rowdesc->GetFirstChild();    // 2. loop to reach the second or nth column  for ( x = 1; x < 2; x ++, coldesc = coldesc->GetNextChild());    // 3. loop to find the desired property tag  colprop = coldesc->GetFirstChild ();  while ( colprop )      if ( stricmp ( colprop->GetNodeName(), "NAME" )           break;    // 4.copy the text contained as child node to the clients buffer  strcpy ( clientbuffer, colprop->GetFirstChild()->GetNodeValue()));        // HOW TO serve the data of third col of the second row    // 1. start with the first ROW in RESULTDATA  rowdata =pStmt->RespRSData->GetFirstChild();    // 2. loop to reach the second or nth row  for ( x = 1; x < 2; x ++,rowdata = rowdata->GetNextChild());    // 3. loop to reach the third col  coldata = rowdata->GetFirstChild ();  for ( y = 1;y  < 3; x ++,coldata = coldata->GetNextChild());    // 4.convert data to type specified by client and copy to clients buffers  \*( int\* )clientbuffer  = atoi ( colprop->GetFirstChild()->GetNodeValue()); |

Similarly, to obtain the actual value of the *mth* column of the *nth*row, we start with the first **ROW** of data, which is the child of**RESULTDATA** and then loop through its siblings using**GetNextSibling** until we reach the desired row. Note again that rows are siblings to one another. After we reach the row, we start with the first column using **GetFirstChild** on the **ROW** and then loop in a fashion similar to what we did for rows until we reach the desired column. The data is in text format since it has been received as an XML stream. The column may be of type float and the client may have asked for it as an integer and specified a buffer of appropriate size. You have to consider all three; i.e., text, float, and integer, and convert it to the destination data type in an optimized way. I have tried to provide the basic conversion in my download samples, but you may need to extend or change this area as per your requirements.

You will find that the architecture of calls and communications fits quite well into the WSDL (Web Services Description Language) specification; see Figure 5.



**Figure 5: The overall view of the ODBC driver and database server.**

These are the latest guidelines in this area of software development (**http://www.w3.org/TR/wsdl**) and are actively supported by companies like Microsoft, Ariba, and IBM.

#### Data Transfer And Fetching Results

ODBC defines a standard way of fetching data from the driver. The application specifies a buffer for obtaining the data immediately or for a deferred fetch. For example, functions like **SQLColAttribute**,**SQLDescribeCol**, and so on, specify a buffer in which the driver puts in the information about columns immediately. On the other hand, **SQLBindCol** specifies a buffer along with a column number, which is stored by the driver in its internal structures associated with the connection and statement. When the application calls**SQLFetch** or any other fetch function, the driver uses the buffer specified using **SQLBindCol** to feed the results. The application has to make sure that it does not free the buffers as long as the driver can use it, and the driver makes sure that it maintains the correct associations with the columns and positions and never frees the buffer. The application also has to specify the length of the buffer so that the driver does not cause a protection fault by exceeding the buffer capacity. Sometimes the driver, because of the context and associated fixed length data type, implicitly knows the length of the buffer. In such cases, the application specified length is ignored.

Since the data to be returned may not be exactly equal to the buffer specified by the application, the driver also needs to return the length of data it is feeding into the buffer and whether to return that any truncation has taken place. The application specifies another fixed length buffer for this, which is typically referred to as length-or-indicator pointer and defined as**Strlen\_Or\_Ind**. It is a pointer of type **SQLLEN** (long data type of C language). The driver either puts in the length of returned data into this buffer or if the data is NULL terminated, the buffer is used as an indicator pointer and fed with a value -3, defined as**SQL\_NTS** in **SQL.H**.

The application also uses the same technique for data it passes to the driver. It can either specify the length explicitly or indicate**NULL** terminated data with **SQL\_NTS**.

ODBC allows the application to specify buffers for multiple rows to be retrieved in a single fetch referred to as rowset buffers. This means that the driver needs to know the exact layout of this buffer for the application to work correctly. The two possible layouts that the application can use are as follows:

* Bind an array to each column. This is called *column-wise binding*because each data structure (array) contains data for a single column.
* Define a structure to hold the data for an entire row and bind an array of these structures. This is called *row-wise binding*because each data structure contains the data for a single row.

The application then calls **SQLBindCol** to bind this array of buffers to columns just as it binds a simple buffer for each single row and column of data. To locate the position of the same column in the next row, the driver moves from its current position either by the size of the column buffer (column wise binding) or by the size of the row buffer (row wise binding). Don’t worry if it is a little confusing at first, most of your clients would not require a block fetch, so you can move on to complete this area a little later in your development cycle. The file GO\_FETCH.CPP in the sample download shows how this is managed.

#### Descriptors — An Important Design Issue

A descriptor handle refers to a data structure that holds information about either columns or dynamic parameters. There are four types of descriptors as defined by ODBC. They are ARD (Application Row Descriptor), APD (Application Parameter Descriptor), IRD (Implementation Row Descriptor,) and IPD (Implementation Parameter Descriptor). Most if not all the IO between the driver, data source, and client takes place via these structures.

ODBC does not specify how you implement these structures or the physical layout of these structures. It says that a particular type of data, for example the column bindings, specified by the application be stored in one structure and referred to as the ARD in your design. The important requirement is that the application should have the ability to use a descriptor handle directly and set the attributes or retrieve data, which in most cases is done implicitly by commonly used functions like **SQLBindCol**,**SQLColAttribute**, etc. The application can allocate the descriptor explicitly using **SQLAllocHandle** or it can obtain the handle descriptor allocated by your driver using **SQLGetStmtAttr**.

Most of the clients will not require direct access to descriptors since no database operation requires that the client gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data, which can be more efficient than calling **SQLBindCol**again, but this is uncommon.

The descriptors are also divided into headers and records. For instance, the binding information for columns specified by the application. The details for each column forms a record in the descriptor, while something common to all columns like binding type goes into the header of ARD. I show in Example 11 the ARD structure as used in my driver.

**Example 11.**

[?](http://drdobbs.com/windows/184416434?pgno=4)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | struct GODBCARDItem {        Word        ColNum;                 // column number        Word        DataConciseType;         // concise data type      Word        DataVerboseType;         // basic data type      Word        DateTimeIntervalCode;    // date-time interval code      Long        DateTimeIntervalPrec;    // date-time precision      Long        NumPrecRadix;    // 2 if approx num type or 10 exact num type      Word        Scale;                 // scale, right of decimal        void\*       DataPtr;                           // col data      Long        DataSize;                          // size of data buffer      Long\*       SizePtr;                           // actual data size      Long\*       SizeIndPtr;                        // size indicator ptr        struct GODBCARDItem\*     Next;     // next col binding      struct GODBCARDItem\*     Prev;     // prev col binding  };    typedef struct GODBCARDItem     GODBCARDITEM;        // ARD-Col data type  typedef struct GODBCARDItem\*    PGODBCARDITEM;       // ARD-Col as pointer type    struct TlyODBCARD {        // header fields      Word           AllocType;              // SQL\_DESC\_ALLOC\_TYPE      ULong         RowArraySize;            // SQL\_DESC\_ARRAY\_SIZE      UWord\*       ArrayStatusPtr;           // SQL\_DESC\_ARRAY\_STATUS\_PTR      Long\*          BindOffsetPtr;          // SQL\_DESC\_BIND\_OFFSET\_PTR      Long           BindTypeOrSize;         // SQL\_DESC\_BIND\_TYPE      Word           DescCount;             // SQL\_DESC\_COUNT        // rows for binding each col      PGODBCARDITEM BindCols;               // col bindings        // container      PGODBCSTMT    Stmt;                  // container statememt  }; |

The one major advantage I found in following this design guide is a more precise framework and easy compliance with more clients. I state this because I initially ignored this and ended up with a complex design and still was unable to support the DTS (Data Transformation Services) module of SQL server for importing data. Finally, I had to redesign the entire thing and centralize all the IO via the descriptor concept. As you develop your own ODBC driver or browse through sample files (GO\_DESC.CPP), you will find many of the calls are simply mapped to the four descriptor functions —**SQLGetDescField/Rec** and **.SQLGetDescField/Rec**.

I will briefly describe the purpose of each type of descriptor. You can find detailed information on layout and content of these structures from the files GODBC.H and GO\_DESC.C in the download samples or on the MSDN site. The application descriptors are supposed to contain information specified by the client application. The application row descriptor therefore contains information about the row as required by the client; say it requires the third column in a buffer of 64 bytes, so this is stored in an ARD record. Similarly, while using parameterized queries, the details of the parameters as specified by the client or obtained from the client should be stored in the APD structures.

The implementation descriptors are supposed to contain data as obtained from the server or derived by the driver itself. Therefore, the result from the server is stored in the IRD structure. Similarly, the parameter related information from the server should be stored in the IPD. Personally, I found the ARD and IRD to be more explicit in terms of their definition and functionality as compared to APD and IPD, which overlap at places.

#### Function Categories

In this section, I provide an overview of the major function categories as required in an ODBC driver. Broadly, the functions can be categorized into: **Connection functions** — SQLConnect, SQLDriverConnect, SQLBrowseConnect, and SQLAllocHandle. I have discussed SQLDriverConnect at reasonable length in the section “A Simple ODBC Client”. **Functions to query the functionality supported by the driver** — SQLGetInfo, SQLGetFunctions, and SQLTypeInfo. SQLGetInfo is a very important function and returns information about the driver and data source associated with a connection. You will need to implement this function in the very beginning in order for clients like MS Word and Excel to work. These clients query quite a few things about your driver before they can extract data using it as a pipe to your DBMS. The function is not very difficult to implement but it becomes a long switch-case statement. Also you have to pick up all possible query attributes from the header files and return an appropriate value either from within the driver or query your DBMS using our XML-SOAP-socket call mechanism. I have a file GO\_INFO.CPP dedicated to this API call. If you use it, you just have to replace the sample values with the correct values for your driver.

SQLGetFunctions returns information about whether a driver supports a specific ODBC function. This function is implemented in the Driver Manager; it can also be implemented in drivers. If a driver implements SQLGetFunctions, the Driver Manager calls the function in the driver. Otherwise, it executes the function itself.

SQLTypeInfo returns information about the data types supported by your driver. This function will be implemented as a call to your server, which will return a resultset with the content as described by the ODBC standard. It is a little confusing when you implement this function since you will be describing the data types in the RESULTDESC, and the data types will come as part of RESULTDATA. Typically the data types describe the data but in this particular API, they are being described. I have provided a possible resultset in the download sample for the dummy server.

**Descriptor Functions** — SQLGetDescField, SQLGetDescRow, SQLSetDescField, and SQLSetDescRow. I have discussed descriptors in the section “Descriptors — an important design issue”.

**Functions to set and retrieve driver attributes** — SQLGetEnvAttr, SQLSetEnvAttr, SQLGetConnAttr, SQLSetConnAttr, SQLGetStmtAttr, and SQLSetStmtattr. As the names imply, these are used to get and set properties at the environment, connection, and statement level.

**Query submission and execution functions** — A query can be executed in a single step using SQLExecDirect or in two steps using SQLPrepare and SQLExecute. The second method is useful when the query is to be executed multiple times maybe with different parameters. The DBMS should be capable of processing and storing it in a form, where it can avoid duplicating some part of the execution process and therefore yield better performance. Queries can be parameterized. I clump these parameter related functions like SQLDescribeParam and SQLParamData also into this category.

**Functions for obtaining metadata about result or datasource**— SQLTables, SQLColumns, SQLProcedures, SQLSpecialColumns, SQLPrecuresColumns, SQLForeignKeys, and SQLPrimaryKeys. These are also known as catalog functions. They help you build a catalog describing the server and database content. Most of the names are self-explanatory. For example, SQLTables is used to collect the tables in a specified database. This function also allows you to collect the names of the databases and users on your server. Similarly, SQLColumns can be used to get the column details of a particular table and SQLProcedure can be used to get the names of the procedures available for execution. You will also need to implement few of these before clients like MS Word and Excel are able to use your driver. I have provided the samples for the one, which are required to make these clients work successfully.

**Functions for obtaining result-data** — Once the driver collects the result from the server, the application can use a number of functions to extract it. Say it can use SQLNumResultCols to get the number of columns in the result. The three important functions for this category are SQLColAttribute, SQLBindCol and SQLFetch. SQLColAttribute is used to get the information about a column in the result; we have used this to get the title in our sample client. SQLBindCol is used to specify the column and buffer associations in which the driver will feed the results. SQLFetch is used to fetch one or more rows into the buffers specified by SQLBindCol.

**Diagnostic functions** — ODBC has a well-defined way of managing diagnostics. As mentioned earlier, a list of messages is to be maintained by the driver at all levels — Environment, Connection, and Statement. The client uses the two functions SQLGetDiagRec and SQLGetDiagField to extract diagnostic information when the function does not return SQL\_SUCCESS. It may not necessarily be an error; the driver can return SQL\_SUCCESS\_WTH\_INFO, which means that there are information messages. For example, if truncation has taken place during fetch, the driver will return this as state: **01004** and message: **String** **data**, **right**, **truncated**. The fields maintained as part of diagnostics include a state code, a message, native error code for your driver, column number, row number, and so on. You will need to have a linked list type structure to maintain the diagnostics messages at every level. The header file GODBC.H shows a possible structure for diagnostics and GO\_DIAG.CPP contains the implementation of these functions as well as functions used internally by the driver to maintain these.

**Clean-up functions** — SQLDisconnect and SQLFreeHandle. SQLDisconnect is used to close a connection available via a connection handle. SQLFreeHandle is used to release any of the handles allocated using SQLAllocHandle.

[**http://msdn.microsoft.com/library/en-us/odbc/htm/odbcodbc\_function\_summary.asp**](http://msdn.microsoft.com/library/en-us/odbc/htm/odbcodbc_function_summary.asp) on MSDN is a good place to start if you are looking for the categorized list of ODBC functions and a brief description.

#### Troubleshooting, Test Tools, Samples, and Resources

**Tracing** — Each call to the ODBC driver manager can be logged along with the parameters and return values. I found this option to be most useful while troubleshooting my driver with clients like MS Word and Excel. There are a number of ways to enable/disable tracing. You can use the Tracing tab on the ODBC administrator from Control Panel and use the button Start Tracing Now. You can also specify the file to which the calls are logged. When the client fails with something like Memory access violation, you can check this log file and precisely locate the call in which the failure occurred or after which the failure occurred. SQLSetConnectAttr to can be used with SQL\_ATTR\_TRACE to enable of disable tracing from within your program.

**DBMON** — This an executable program provided on the windows platform as part of Platform SDK. It allows you to track your program using the OutputDebugString Windows API. The DBMON program runs as a separate process in a console window and does not show any output by itself. The string you output using OutputDebugString from within your program or DLL is immediately visible on the DBMON window. This is especially useful in debugging DLLs. The skeleton driver we developed uses this API for tracking calls to the ODBC functions within our driver.

**Your own log** — I strongly suggest that you create your own log mechanism in your driver. Entry to every call is logged along with the parameter details. I suggest this in addition to the Driver Manager log since there are situations where the driver manager does some intermediate processing or changes the call name to due to version issues or even calls a number of functions from your driver on a single call from the client. Besides in cases of abnormal termination, you also need to know whether the failure occurred on an instruction within your driver code, driver manager code, or the application code. I found this very useful in tracking the errors much faster. There was also a situation where the driver manager was failing to call my function although I had exported it. The driver manager showed a log that indicated that it was calling my function but there was a prototype and calling convention mismatch and it was failing to do so.

**Test clients** — Microsoft also provides test clients like ODBCTE32.EXE to test your driver. You can select the calls and parameter values for each call and see what your driver is returning or executing. Once you are comfortable with test clients like these, you can switch over to Microsoft Query, which is the tool used by most of the other Office components like Word and Excel to get data using your driver. MSQRY32.Exe comes as a part of MS Office.

**DLL and LIBs** — You must make sure that calling convention for your functions is set correctly to stdcall. This is important or else name mangling and stack-cleanup issues will cause the driver manager or client to fail in calling functions from your driver.

**Diagnostic functions** — If the function fails with an error, the driver manager is supposed to return the details of the errors via the diagnostic functions. Ideally your driver should also maintain and return complete details of the error as discussed in the previous section.

I have covered the major concepts with code snippets about how you can convert our skeleton driver into a real life driver. The complete source code is available as download, and you can use the associated readme.txt file to test it out along with the dummy server. Once you are clear with the concepts, I believe it’s an issue of coding it all in a week or two. Also I hope that this article will save you from many trial and error situations I faced while developing my driver.

#### Microsoft Office or Star Office

Your clients can use MS Word directly to mail merge forms and letters picking up data from the server just by using point and click. They can now prepare a balance sheet as per their format or statutory guidelines using the formatting capabilities of Excel and raw data from your server. The possibilities aren’t limited to Microsoft products. You can access your data in Star Office as well.

The option in MS Excel 2000 is on the Tools menu - Get External Data - New Database Query and from then on, it’s a normal ODBC query. Similarly, the Mail Merge option is available on the Tools menu in MS Word 2000. You can even do ADO programming and extract data using VB or VBA.

#### Conclusion

One of the requirements for ODBC was that a single application binary had to work with multiple DBMSs. It is for this reason that ODBC does not use embedded SQL or module languages, each of which is tied to DBMS-specific precompilers. Thus, applications must be recompiled for each DBMS and the resulting binaries work only with a single DBMS. It is a logistical nightmare to deliver multiple versions of high-volume, shrink-wrapped software to customers. Also, personal computer applications often need to access multiple DBMSs simultaneously.

On the other hand, a call-level interface can be implemented through libraries, or database drivers, that reside on each local machine; a different driver is required for each DBMS. Because modern operating systems can load such libraries at run time, a single application can access data from different DBMSs without recompilation and can also access data from multiple databases simultaneously. As new database drivers become available, users can just install these on their computers without having to modify, recompile, or relink their database applications.

The architecture we have used for IPC/RPC makes your database easily accessible across a WAN also. You need not have to go through the extensive plumbing and sophisticated configuration and compatibility issues required for COM or CORBA.

You are open to the world without much change to your internal architecture. The system for which I developed an ODBC driver is not a typical RDBMS system and is highly optimized for our kind and scale of application, but we were able to expose it in the form of an RDBMS by adding a thin layer without much difficulty.

With this article, I hope that somebody trying to develop an ODBC driver will not be restricted to only two options — the very fat ODBC DDK from Microsoft or an expensive consultancy from companies specializing in this area.

**Vikash K Agarwal***is working as Technology Architect for Tally Solutions, Bangalore. He’s worked for over 10 years in software development with C as the prime focus. Vikash has also worked in and with VC++, Delphi, VB, COM, PHP, ASP, MYSQL, SQL Server, Lex and Yacc, Sockets, XML, and PKI. When not making software work, he ponders on what makes the universe work. He can be contacted at*[*vikash\_agarwal@hotmail.com*](mailto:vikash_agarwal@hotmail.com)*.*

<http://drdobbs.com/windows/184416434>