**A Quick Guide for SQL Server Native Client OLE DB to ODBC Conversion**

**Introduction**

OLE DB and ODBC are application programming interfaces (APIs) designed to provide access to a wide range of data sources. Microsoft implemented these interfaces for SQL Server through a SQL Server Native Client (SNAC) OLE DB provider and a SNAC ODBC driver. A C/C++ application can access a SQL Server database through these interfaces. The ODBC API is a set of flexible and powerful, but straightforward functions while OLE DB API consists of a set of object oriented interfaces based on COM.

ODBC is designed to provide access primarily to SQL data in a multiplatform environment. OLE DB is designed to provide access to all types of data in an Component Object Model (COM) environment. OLE DB includes the Structured Query Language (SQL) functionality defined in ODBC but also defines interfaces suitable for gaining access to data other than SQL data. For more background on ODBC and OLE DB please refer to [Microsoft Open Database Connectivity](http://msdn.microsoft.com/en-us/library/ms710252%28v%3DVS.85%29.aspx) and [Microsoft OLE DB](http://msdn.microsoft.com/en-us/library/ms722784%28VS.85%29.aspx) on MSDN.

With the recent announcement of SNAC OLE DB provider deprecation, some existing SNAC OLE DB applications will need to be migrated to ODBC in the future. It is worth mentioning that the SNAC OLE DB provider deprecation only affects those applications that leverage SNAC OLE DB provider to access SQL Server data. Other OLE DB applications that use OLE DB API to access other data sources through OLE DB providers other than SNAC OLE DB provider, such as accessing ACE provider to Microsoft Office data, are not affected. This paper provides an introduction for those not familiar with the ODBC API on how to convert an OLE DB application to an ODBC application. This paper’s scope is limited to discussing functionality conversions only. Other concerns, such as performance comparisons between OLE DB and ODBC will be addressed in future papers.

**ODBC Overview**

Open Database Connectivity (ODBC) is a widely accepted application programming interface (API) for database access. It is based on the Call-Level Interface (CLI) specifications from Open Group and ISO/IEC for database APIs and uses Structured Query Language (SQL) as its database access language.

ODBC is designed for maximum interoperability - that is, the ability of a single application to access different database management systems (DBMSs) with the same source code. Database applications call functions in the ODBC interface, which are implemented in database-specific modules called drivers. 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. Because drivers are loaded at run time, a user only has to add a new driver to access a new DBMS; it is not necessary to recompile or relink the application.

The paragraphs below explain some of the major characteristics of ODBC API:

* **ODBC is a call-level interface.** To solve the problem of how applications access multiple DBMSs using the same source code, ODBC defines a standard CLI. This contains all of the functions in the CLI specifications from Open Group and ISO/IEC and provides additional functions commonly required by applications.

A different library, or driver, is required for each DBMS that supports ODBC. The driver implements the functions in the ODBC API. To use a different driver, the application does not need to be recompiled or relinked. Instead, the application simply loads the new driver and calls the functions in it. To access multiple DBMSs simultaneously, the application loads multiple drivers. How drivers are supported is operating system–specific. For example, on the Microsoft® Windows® operating system, drivers are dynamic-link libraries (DLLs).

* **ODBC defines a standard SQL grammar.** In addition to a standard call-level interface, ODBC defines a standard SQL grammar. This grammar is based on the Open Group SQL CAE specification. Differences between the two grammars are minor and primarily due to the differences between the SQL grammar required by embedded SQL (Open Group) and a CLI (ODBC). There are also some extensions to the grammar to expose commonly available language features not covered by the Open Group grammar.

Applications can submit statements using ODBC or DBMS-specific grammar. If a statement uses ODBC grammar that is different from DBMS-specific grammar, the driver converts it before sending it to the data source. However, such conversions are rare because most DBMSs already use standard SQL grammar.

* **ODBC provides a Driver Manager to manage simultaneous access to multiple DBMSs.** Although the use of drivers solves the problem of accessing multiple DBMSs simultaneously, the code to do this can be complex. Applications that are designed to work with all drivers cannot be statically linked to any drivers. Instead, they must load drivers at run time and call the functions in them through a table of function pointers. The situation becomes more complex if the application uses multiple drivers simultaneously.

Rather than forcing each application to do this, ODBC provides a Driver Manager. The Driver Manager implements all of the ODBC functions — mostly as pass-through calls to ODBC functions in drivers — and is statically linked to the application or loaded by the application at run time. Thus, the application calls ODBC functions by name in the Driver Manager, rather than by pointer in each driver.

When an application needs a particular driver, it first requests a connection handle with which to identify the driver and then requests that the Driver Manager load the driver. The Driver Manager loads the driver and stores the address of each function in the driver. To call an ODBC function in the driver, the application calls that function in the Driver Manager and passes the connection handle for the driver. The Driver Manager then calls the function by using the address it stored earlier.

In Windows, the Driver Manager is part of the core operating system. It is installed by default on every Windows machine as part of the ODBC32.DLL library.

* **ODBC exposes a significant number of DBMS features but does not require drivers to support all of them.** ODBC exposes a significant number of features — more than are supported by most DBMSs — but requires drivers to implement only a subset of those features. Drivers implement the remaining features only if they are supported by the underlying DBMS or if they choose to emulate them. Thus, applications can be written to exploit the features of a single DBMS as exposed by the driver for that DBMS, to use only those features used by all DBMSs, or to check for support of a particular feature and react accordingly.

An application can determine what features a driver and DBMS support, ODBC provides two functions (**SQLGetInfo** and **SQLGetFunctions**) that return general information about the driver and DBMS capabilities and a list of functions the driver supports. ODBC also defines API and SQL grammar conformance levels, which specify broad ranges of features supported by the driver. For more information, see [Conformance Levels](http://msdn.microsoft.com/en-us/library/ms716468%28v%3DVS.85%29.aspx).

It is important to remember that ODBC defines a common interface for all of the features it exposes. Because of this, applications contain feature-specific code, not DBMS-specific code, and can use any drivers that expose those features. One advantage of this is that applications do not need to be updated when the features supported by a DBMS are enhanced; instead, when an updated driver is installed, the application automatically uses the features because its code is feature-specific, not driver-specific or DBMS-specific.

ODBC has gone through several revisions in the past. The most current version is ODBC 3.8. Since the release of ODBC 3.0 in 1995, it has aligned with the following specifications and standards that deal with the Call-Level Interface (CLI). (The ODBC features are a superset of each of these standards.)

* The Open Group CAE Specification "Data Management: SQL Call-Level Interface (CLI)"
* ISO/IEC 9075-3:1995 (E) Call-Level Interface (SQL/CLI)

An application written to the Open Group and ISO CLI specifications will work with an ODBC 3.x driver through an ODBC 3.x Driver Manager. In addition to the call-level interface specifications contained in the ISO/IEC and Open Group CLI standards, ODBC implements the following features. (Some of these features existed in versions of ODBC prior to ODBC 3.x.)

* Multi-row fetches by a single function call
* Binding to an array of parameters
* Bookmark support including fetching by bookmark, variable-length bookmarks, and bulk update and delete by bookmark operations on non-contiguous rows
* Column-wise and row-wise binding
* Binding offsets
* Support for batches of SQL statements, either in a stored procedure or as a sequence of SQL statements executed through **SQLExecute** or **SQLExecDirect**
* Exact or approximate cursor row counts
* Positioned update and delete operations and batched updates and deletes by function call (**SQLSetPos**)
* Catalog functions that extract information from the information schema without the need for supporting information schema views
* Escape sequences for outer joins, scalar functions, date/time literals, interval literals, and stored procedures
* Code-page translation libraries
* Reporting of a driver's ANSI-conformance level and SQL support
* On-demand automatic population of implementation parameter descriptor
* Enhanced diagnostics and row and parameter status arrays
* Datetime, interval, numeric/decimal, and 64-bit integer application buffer types
* Asynchronous execution
* Stored procedure support, including escape sequences, output parameter binding mechanisms, and catalog functions
* Connection enhancements including support for connection attributes and attribute browsing

The current SNAC ODBC driver is a 3.x driver. For the complete list of current SNAC ODBC APIs, please see [ODBC Programmer's Reference](http://msdn.microsoft.com/en-us/library/ms714177%28v%3DVS.85%29.aspx).

**Feature Differences between OLE DB and ODBC**

Before attempting to convert an OLE DB application to ODBC, one must first consider the high level architecture and feature differences. The following is a summary of these differences and how they might impact the conversion.

For the rest of this document, ODBC refers to the current SNAC ODBC driver (SNAC 10.x) and Driver Manager shipped with Windows 7. OLE DB refers to the current SNAC OLE DB provider (SNAC 10.x). combination of SNAC 10.x ODBC driver and Driver Manager in Windows 7 contain many SQL Server specific features.

* ODBC applications invoke ODBC APIs through an ODBC driver manager. This allows applications to use different drivers without recompilation. The application developers should be aware that the level of ODBC functionality depends not only on the driver but also the Driver Manager. The Windows Driver Manger ships with the operating system.
* COM-based OLE DB interfaces are reference counted . ODBC handles, on the other hand, are completely under the programmers’ control. One can easily use smart pointer classes to achieve reference counting, if desired. The application programmers manage the ODBC handles and manages their life cycle.
* There is no direct equivalent for OLE DB enumerators in ODBC. However, ODBC provides SQLDataSources which returns various pieces of information about a data source.
* There is no direct equivalent for OLE DB generic ITableDefinition, IOpenRowset, IRowsetUpdate interfaces, which allows direct data retrieval without requiring the user to use specific Transact-SQL statements, in ODBC. However, it is easy to retrieve equivalent data using ODBC APIs by supplying the appropriate Transact-SQL statements.
* The set of initialization properties is dynamic and discoverable in OLE DB via IDBProperties::GetPropertyInfo. In ODBC the properties are predefined. It is rare that the initialization properties are set in an arbitrary manner by the end user, and prior knowledge of the predefined and therefore tested ODBC attributes provide a more straightforward and reliable way of initialization.
* OLE DB properties can be optional or required and the OLE DB provider tries to provide the best match based on whether the option is required or optional. ODBC provides more straightforward and simple model that does not allow some complicated (and therefore confusing) overriding scenarios. For instance it is much easier to specify a specific cursor type in ODBC than in OLE DB.
* There is no direct equivalent of OLE DB IRowsetFind interface for finding specific rows in a rowset on the client. However the same functionality can be achieved with a properly formulated Transact-SQL request in ODBC. For relational data source such as SQL Server, it is sometimes more desirable to let the server to select the right rows because of the sophistication of server side optimization.
* ODBC uses application memory to hold query results and OLE DB uses provider-owned memory. Using application memory is more straightforward and easier to understand. Using provider owned memory results in more complicated memory management logic (e.g. using CoTaskMemFree).
* OLE DB creates implicit connections in order to be able to execute multiple commands from a single session when processing is not finished for a specific result set. This behavior is not always obvious from end user perspective and entails an overhead of establishing a new connection which can’t be pooled and may cause problems with transaction enlistments. ODBC does not create implicit connections when handling multiple commands, instead it shares a single connection by leveraging the Multiple Active Result Sets (MARS) feature.
* OLE DB metadata is provided with schema rowsets. ODBC metadata is provided with APIs like SQLGetDescField and SQLGetDescRecord.
* There is a set of SQL Server specific APIs for bulk copy operations (BCP) in ODBC. To use this set of APIs one must directly link to the specific SNAC driver dll (instead of the driver manger dll).
* ODBC supports conversions for native C/C++ primitive types as well as some composite types using structures. There is no direct support of converting OLE DB types to native C/C++ types.
* Both OLE DB and ODBC support asynchronous processing. OLE DB asynchronous processing is notification based and ODBC asynchronous processing is polling based. ODBC will support notification based asynchronous processing in the near future.
* OLE DB provides streaming support implementing ISequentialStream. ODBC does not support streaming per se but one can achieve similar objectives with chunking using SQLGetData and SQLPutData.
* The OLE DB provider provides a data type compatibility knob, for down-leveling newer database types. There is no direct support of this in the SNAC ODBC.
* OLE DB provides a simplified RPC syntax (RPC escape sequence). There is no direct equivalent in ODBC, but ODBC supports both canonical calls and T-SQL EXEC.

**Mapping Basic OLE DB Objects to ODBC APIs**

In this section we exam the basic OLE DB objects and attempt to map the functionality of these objects to the corresponding ODBC APIs. In the next section we will look at how the mapping can be extended to cover an entire typical application. We will look at a typical end-to-end application flow and see how the application is implemented in ODE DB and how it can be converted to use the corresponding ODBC APIs.

#### Data Source Object

The data source object is the initial object instantiated by calling the OLE function **CoCreateInstance** with a given OLE DB data provider's unique class identifier (CLSID).

The corresponding ODBC concept is captured as part of an ODBC environment and an ODBC connection. There is no separate structure that represents a data source in ODBC. An environment is a global context in which to access data; associated with an environment is any information that is global in nature, such as:

* The environment's state
* The current environment-level diagnostics
* The handles of connections currently allocated on the environment
* The current settings of each environment attribute

An ODBC environment is created by calling **SQLAllocHandle**. ODBC environment determines the behaviors of subsequent ODBC calls (via environment attributes) such as how connection pooling works, which version of the ODBC APIs to use, and whether string data returned is NULL terminated. The environment attributes can be changed by calling **SQLSetEnvAttr**. An ODBC environment is required for several other ODBC calls dealing with global operations. The most common operation is to create new ODBC connections.. In OLE DB, a Data Source Object stores information about which database server to target. In ODBC, this information is stored in an ODBC connection. ODBC connections are explained in more detail in the next paragraph.

#### Session Object

A session object defines the scope of a transaction and generates rowsets from the data source. If the provider supports commands, the session also acts as a command factory. The data source object can also support interfaces for describing schema information and for creating tables and indexes for providers that support that functionality.

Connections are the ODBC equivalent of OLE DB Session Objects. An ODBC connection is a structure that conceptually captures the information regarding a data channel to a database. A connection consists of a driver and a data source. The connection defines not only which driver to use but which data source to use with that driver. A connection contains the following:

* The state of the connection
* The current connection-level diagnostics
* The handles of statements and descriptors currently allocated on the connection
* The current settings of each connection attribute

A live ODBC connection is created by first calling **SQLAllocHandle**, passing in a handle to an ODBC environment, to create a connection handle and then calling one of the following API calls to actually open the data channel to a data source for a user: **SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**.

#### Command Object

If a provider supports building and executing queries, it exposes a command object. A command object is generated from a session object. It is used to specify, prepare, and execute a DML query or DDL definition and associated properties.

The corresponding concept in ODBC is ODBC statement. An ODBC statement captures the necessary information for a SQL query execution. It contains the following information:

* The state of the statement
* The current statement-level diagnostics
* The handles of various structures such as the parameter bindings and the query execution result.
* The current settings of each statement attribute

A statement is created by calling **SQLAllocHandle** and passing in a connection handle. Once the statement is created, one can set the SQL query to be executed, set various statement attributes, bind parameters, or prepare and execute the SQL query by calling various ODBC APIs passing in the statement handle as a parameter.

#### Rowset Object

A rowset object is a shared data object that represents tabular data, such as a result set returned by executing a query. If the provider supports commands, rowsets are used to represent the results of row-returning queries. There are a number of other methods in OLE DB, such as the schema functions, that return information in the form of a rowset.

The corresponding concept in ODBC is ODBC result set. ODBC result set is not a data structure that can exist independently. Rather, it is part of the statement and can only be accessed via ODBC calls on a statement handle. When an ODBC statement is executed one or more result sets are attached to the statement. These result sets can then be accessed through ODBC API calls such as **SQLMoreResults**, **SQLFetch, SQLBindCol,** and **SQLGetData**.

### Other Objects

The following objects are also defined in OLE DB. They provide recursive data source enumeration, enhanced transaction control, and extended error retrieval.

##### Enumerator Object

Enumerator objects list the data sources and enumerators visible to that enumerator. This is similar to the information provided by **SQLDataSources**, except that the information can be recursive.

##### Transaction Object

In addition to supporting **ITransactionLocal** on the session, providers that support transactions can optionally support the creation of a transaction object.

In ODBC transaction are managed via

* Call the **SQLSetConnectAttr** function with *Attribute* set to SQL\_ATTR\_AUTOCOMMIT and *ValuePtr* set to SQL\_AUTOCOMMIT\_OFF to start implicit transaction mode.
* The connection remains in implicit transaction mode until you call **SQLSetConnectAttr** with *Attribute* set to SQL\_ATTR\_AUTOCOMMIT and *ValuePtr* set to SQL\_AUTOCOMMIT\_ON.
* Call the **SQLEndTran** function with *CompletionType* set to either SQL\_COMMIT or SQL\_ROLLBACK to commit or roll back each transaction.

#### Error Object

In addition to the return codes and status information returned by each method in OLE DB, providers can optionally expose an OLE DB error object for extended error information, such as a description of the error or the appropriate SQLSTATE. This is similar to the information returned by **SQLGetDiagRec**.

**Converting a Basic SNAC OLE DB Application to SNAC ODBC**

The application flow in ODBC is similar to the application flow in OLE DB. In both cases, the application:

1. Initializes the environment.
2. Connects to a data source.
3. Creates and executes a command.
4. Processes results, if any.
5. Cleans up.

A typical OLE DB application achieves the above steps by:

1. Initializes COM.
2. Connects to a data source object.
3. Creates and executes a command.
4. Processes the results.
5. Releases objects and uninitializes COM.

In an ODCB application the corresponding steps are:

1. Allocates an environment handle.
2. Allocates a connection handle and connects to a data source.
3. Creates and executes a statement.
4. Processes the results.
5. Disconnects & releases various handles.

**Initializing the Environment**

In OLE DB, initialization of the environment is achieved by a call to **CoInitializeEx**, which initializes the COM library. After the COM library is initialized, the application can use **CoCreateInstance** to either instantiate provider object directly using its class ID or to instantiate OLE DB Core Services (CLSID\_MSDAINITIALIZE).

In ODBC, environment consists of the ODBC driver manager and the ODBC environment handle. The application generally dynamically links to the ODBC Driver Manager (Odbc32.dll) and therefore there is no explicit action on the part of application required in order to load it (except start making ODBC calls). Once loaded Driver Manager loads and directs calls to the appropriate driver. The Driver manager is responsible for connection pooling serving the similar purpose as OLE DB Core Services does.

The ODBC environment handle is created by the application via the ODBC call **SQLAllocHandle**. This environment handle is then used to control global behaviors of the ODBC APIs such as whether connection pooling is enabled or not.

|  |  |
| --- | --- |
| **OLE DB** | **ODBC** |
| CHECKED(CoInitializeEx(NULL, COINIT\_MULTITHREADED)); CHECKED(CoCreateInstance(CLSID\_MSDAINITIALIZE, NULL, CLSCTX\_INPROC\_SERVER, IID\_IDataInitialize, (void\*\*)&apIDI)); | CHECKED(SQLAllocHandle(SQL\_HANDLE\_ENV,SQL\_NULL\_HANDLE, &henv));CHECKED(SQLSetEnvAttr( henv, SQL\_ATTR\_ODBC\_VERSION, (void \*)SQL\_OV\_ODBC3, 0 )); |

**Connecting to a Data Source**

The data source object exposes the **IDBInitialize** and **IDBProperties** interfaces that contain the methods to connect to a data source. The authentication information such as user ID, password, and the name of the data source are specified as properties of the data source object by calling **IDBProperties::SetProperties.**

Alternatively **IDataInitailize::GetDataSource** interface provided by OLE DB Core Services allows using a connection string and the corresponding property sets will be created from the connection string.

The method **IDBInitialize::Initialize** uses the specified properties to connect to the data source.

In ODBC, establishing a connection involves the following steps:

1. Call **SQLAllocHandle** to allocate a connection handle.
2. Build a connection string containing keywords for authentication information, such as user ID, password, and the name of the data source.
3. Call **SQLDriverConnect,** providing the connection string and other information, such as level of prompting and the application's window handle where appropriate.

Properties are used in OLE DB to specify options, such as initialization information on the data source object or supported properties of a rowset, as well as to discover properties of certain objects, such as the updatability of a rowset.

Corresponding attributes to control these behaviors are spread among the environment, connection, and statement attributes in ODBC. These attributes can be get/set by the ODBC APIs **SQLGetEnvAttr/SQLSetEnvAttr**, **SQLGetConnectAttr/SQLSetConnectAttr**, **SQLGetStmtAttr/SQLSetStmtAttr**.

|  |  |
| --- | --- |
| **OLE DB** | **ODBC** |
| CHECKED(apIDI->GetDataSource(NULL, CLSCTX\_INPROC\_SERVER, L"Provider=SQLNCLI11; Integrated Security=SSPI; Data Source=.;", IID\_IDBInitialize, (IUnknown\*\*)&apIDBInit));CHECKED(apIDBInit->Initialize()); | CHECKED(SQLAllocHandle(SQL\_HANDLE\_ENV,SQL\_NULL\_HANDLE, &henv));CHECKED(SQLSetEnvAttr( henv, SQL\_ATTR\_ODBC\_VERSION, (void \*)SQL\_OV\_ODBC3, 0 ));CHECKED(SQLAllocHandle(SQL\_HANDLE\_DBC, henv, &hdbc));CHECKED(SQLDriverConnectW ( hdbc, NULL, wszConnStr, SQL\_NTS, NULL, 0, &pcbConnStrOut, SQL\_DRIVER\_NOPROMPT )); |

**Creating and Executing a Command**

In OLE DB the data source object exposes the **IDBCreateSession** interface through which a session object can be created. A session defines transaction scope and acts mainly as a command generator by supporting the **IDBCreateCommand** interface. Commands contain a DML query or a DDL definition. The execution of a row-returning command yields a rowset object.

In ODBC the connection handle is used for establishing connections as well as scoping transactions, so the application must allocate and connect a separate connection handle for each concurrent transaction. In OLE DB you can have multiple session objects on one initialized data source object, which means you can have multiple concurrent transactions without having to make multiple connections (where necessary, the provider makes additional connections using the connection information provided in the initialization of the data source object).

The command object in OLE DB is similar to the ODBC statement handle in the unexecuted state. An ODBC connection can have several statement handles. An ODBC application performs the following steps to execute a command:

1. Calls **SQLAllocHandle** to allocate a statement.
2. Calls **SQLSetStmtAttr** to set various attributes that affect how the command is executed (such as query time-out) and how the cursor is opened (such as scrollability, updatability, and so on).
3. Calls **SQLPrepare** if it wants to prepare the statement for repeated execution.
4. Calls **SQLExecute** or **SQLExecDirect** to execute the query.

|  |  |
| --- | --- |
| **OLE DB** | **ODBC** |
| CHECKED(apIDBCreateSession->CreateSession(NULL, IID\_IDBCreateCommand, (IUnknown\*\*)&apIDBCreateCommand));CHECKED(apIDBCreateCommand->CreateCommand(NULL, IID\_ICommandText, (IUnknown\*\*)&apCommand));CHECKED(apCommand->SetCommandText(DBGUID\_DBSQL, L"{call sp\_who(?)}"));CHECKED(apCommand->QueryInterface(&apICommandPrepare));CHECKED(apICommandPrepare->Prepare(0));CHECKED(apCommand->Execute( NULL, IID\_IRowset, &dbParam, NULL, (IUnknown\*\*)&apIRowset));  | CHECKED(SQLAllocHandle(SQL\_HANDLE\_STMT, hdbc, &hstmt));CHECKED(SQLPrepareW(hstmt,L"{call sp\_who(?)}", SQL\_NTS)); CHECKED(SQLExecute(hstmt)); |

**Processing Results**

In OLE DB a rowset provides a standard way to work with a multiset of rows where each row contains one or more columns of data. This provides a standard way for all OLE DB data providers to expose data in tabular form.

Conceptually, rowsets are similar to result sets in ODBC; their implementation, however, is different.

The OLE DB rowset basically takes the memory buffer out of the application and puts it in a stand-alone, shared data object and components access data in this shared memory through high-performance binding descriptions known as *accessors*.

In ODBC, when the application calls **SQLFetch** or **SQLGetData** to retrieve data, the data is read from the database into the application's memory. At that point, the application owns the data; neither the ODBC driver nor other components have access to that data.

The main differences between how data is retrieved in ODBC and how data is retrieved in OLE DB are a direct result of the differences between the application-owned data model of ODBC and the shared-data model of OLE DB.

The metadata about the query result is obtained in OLE DB via the interfaces:

* **IColumnsInfo**. Provides information about the columns of the rowset (metadata). This is similar to **SQLDescribeCol** in ODBC.
* **IRowsetInfo**. Provides information about the rowset. This is similar to **SQLGetStmtAttr** in ODBC.

Same set of metadata can be obtained via the following ODBC APIs

* **SQLNumResultCols** which provides the number of columns in the result
* **SQLDescribeCol** and **SQLColAttribute** which provides information about the columns of the result.
* **SQLGetStmtAttr which provides access for the following statement attributes.**
	+ SQL\_ATTR\_APP\_ROW\_DESC
	+ SQL\_ATTR\_IMP\_ROW\_DESC
	+ SQL\_ATTR\_ROW\_BIND\_OFFSET\_PTR
	+ SQL\_ATTR\_ROW\_OPERATION\_PTR

In OLE DB rows in the result are fetched via **IRowSet** interface in conjunction with **IAccessor**. In ODBC to retrieve a row of data from the result set in ODBC, the application:

1. Calls **SQLBindCol** to bind the columns of the result set to storage locations, if not already done.
2. Calls **SQLFetch** to move to the next row and retrieve data for all bound columns.
3. Calls **SQLGetData** to retrieve data from unbound columns.

|  |  |
| --- | --- |
| **OLE DB** | **ODBC** |
| while (hr == S\_OK){ DBCOUNTITEM cRows = 0; CHECKED(apIRowset->GetNextRows( DB\_NULL\_HCHAPTER, 0, 1, &cRows, &rghRows)); if (cRows) { CHECKED(apIRowset->GetData(rgRow[0], hRowAccessor, &rowData)); if (hr == S\_OK) { wprintf(L"%d\t%s\t%s\n", rowData.spid, rowData.status, rowData.hostName); } CHECKED(apIRowset->ReleaseRows( cRows, rghRows, NULL, NULL, NULL)); }} | #ifdef USE\_SQLBindColCHECKED(SQLBindCol( hstmt, 1, SQL\_C\_LONG, &rowData.spid, sizeof(rowData.spid), cbLen));CHECKED(SQLBindCol( hstmt, 3, SQL\_C\_WCHAR, &rowData.status, sizeof(rowData.status), cbLen+1));CHECKED(SQLBindCol( hstmt, 5, SQL\_C\_WCHAR, &rowData.hostName, sizeof(rowData.hostName), cbLen+2));#endifwhile(SQL\_SUCCEEDED(rc = SQLFetch (hstmt))){#ifndef USE\_SQLBindCol CHECKED(SQLGetData( hstmt, 1, SQL\_C\_LONG, &rowData.spid, sizeof(rowData.spid), cbLen)); CHECKED(SQLGetData( hstmt, 3, SQL\_C\_WCHAR, &rowData.status, sizeof(rowData.status), cbLen+1)); CHECKED(SQLGetData( hstmt, 5, SQL\_C\_WCHAR, &rowData.hostName, sizeof(rowData.hostName), cbLen+2));#endif wprintf(L"%d\t%s\t%s\n", rowData.spid, rowData.status, rowData.hostName);} |

**Cleaning Up**

OLE DB interfaces are reference counted and the objects are deleted when the reference count goes to zero. In ODBC when an application has finished using a data source, it calls **SQLDisconnect**. **SQLDisconnect** frees any statements that are allocated on the connection and disconnects the driver from the data source. The handles should be freed by calling **SQLFreeHandle**.

|  |  |
| --- | --- |
| **OLE DB** | **ODBC** |
| // Using autopointers therefore no special code is necessary | if (hstmt){ SQLFreeHandle(SQL\_HANDLE\_STMT, hstmt); hstmt = NULL;}if (hdbc){ SQLDisconnect(hdbc); SQLFreeHandle(SQL\_HANDLE\_DBC, hdbc); hdbc = NULL;}if (henv){ SQLFreeHandle(SQL\_HANDLE\_ENV, henv); henv = NULL;} |

**Appendix A: Sample OBDC Application**

#include <windows.h>

#include "sql.h"

#include "sqlext.h"

#include "stdio.h"

#define SECURITY\_WIN32

#include <security.h> // GetUserNameEx

#define CHECKED(\_rc) do {if(SQL\_ERROR==(rc=\_rc)){PrintRC(rc); throw;}}while(0)

#define CHECKED\_BOOLEAN(b) do {if(!b){ShowError(); throw;}}while(0)

#define PrintRC(rc) PrintRC\_internal(rc, \_\_FILE\_\_, \_\_LINE\_\_)

void PrintRC\_internal(SQLRETURN rc, char \* szFile, ULONG uLine)

{

 wprintf(L"Error RC = %d (0x%x) %hs(%d)\n", rc, rc, szFile, uLine);

}

void ShowError()

{

 // Retrieve the system error message for the last-error code

 LPVOID lpMsgBuf;

 DWORD dw = GetLastError();

 DWORD len=0;

 len=FormatMessageW(

 FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |

 FORMAT\_MESSAGE\_FROM\_SYSTEM |

 FORMAT\_MESSAGE\_IGNORE\_INSERTS,

 NULL,

 dw,

 MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),

 (LPWSTR) &lpMsgBuf,

 0, NULL );

 if (!len)

 {

 wprintf(L"Error from FormatMessage %d\n Original error %d\n", GetLastError(), dw);

 }

 else

 {

 wprintf(L"(%d) %s\n", dw, (WCHAR\*)lpMsgBuf);

 }

 LocalFree(lpMsgBuf);

}

// Comment out the following line to use SQLGetData instead of binding columns before fetch.

#define USE\_SQLBindCol

int \_\_cdecl main()

{

 HENV henv = NULL;

 HDBC hdbc = NULL;

 HSTMT hstmt = NULL;

 SWORD pcbConnStrOut = 0;

 WCHAR wszUserName[MAX\_PATH];

 DWORD dwLength = MAX\_PATH;

 SQLINTEGER cbNTS = SQL\_NTS;

 SQLRETURN rc;

 WCHAR \* wszConnStr = L"SERVER=.;DRIVER={SQL Server Native Client 11.0};Trusted\_Connection=yes;";

 try

 {

 SQLLEN cbLen[3];

 struct sRowData

 {

 int spid;

 WCHAR status[60];

 WCHAR hostName[MAX\_PATH];

 } rowData;

 CHECKED(SQLAllocHandle(SQL\_HANDLE\_ENV,SQL\_NULL\_HANDLE, &henv));

 CHECKED(SQLSetEnvAttr( henv, SQL\_ATTR\_ODBC\_VERSION, (void \*)SQL\_OV\_ODBC3, 0 ));

 CHECKED(SQLAllocHandle(SQL\_HANDLE\_DBC, henv, &hdbc));

 CHECKED(SQLDriverConnectW

 (

 hdbc,

 NULL,

 wszConnStr,

 SQL\_NTS,

 NULL,

 0,

 &pcbConnStrOut,

 SQL\_DRIVER\_NOPROMPT

 ));

 CHECKED(SQLAllocHandle(SQL\_HANDLE\_STMT, hdbc, &hstmt));

 CHECKED(SQLPrepareW(hstmt,L"{call sp\_who(?)}", SQL\_NTS));

 // Get the user name from the OS:

 CHECKED\_BOOLEAN(GetUserNameExW(NameSamCompatible, wszUserName, &dwLength));

 CHECKED(SQLBindParameter(

 hstmt,

 1,

 SQL\_PARAM\_INPUT,

 SQL\_C\_WCHAR,

 SQL\_WCHAR,

 MAX\_PATH,

 0,

 wszUserName,

 MAX\_PATH,

 &cbNTS));

 CHECKED(SQLExecute(hstmt));

 wprintf(L"spid\tstatus\t\t\t\thostname\n");

#ifdef USE\_SQLBindCol

 CHECKED(SQLBindCol(

 hstmt,

 1,

 SQL\_C\_LONG,

 &rowData.spid,

 sizeof(rowData.spid),

 cbLen));

 CHECKED(SQLBindCol(

 hstmt,

 3,

 SQL\_C\_WCHAR,

 &rowData.status,

 sizeof(rowData.status),

 cbLen+1));

 CHECKED(SQLBindCol(

 hstmt,

 5,

 SQL\_C\_WCHAR,

 &rowData.hostName,

 sizeof(rowData.hostName),

 cbLen+2));

#endif

 while(SQL\_SUCCEEDED(rc = SQLFetch (hstmt)))

 {

#ifndef USE\_SQLBindCol

 CHECKED(SQLGetData(

 hstmt,

 1,

 SQL\_C\_LONG,

 &rowData.spid,

 sizeof(rowData.spid),

 cbLen));

 CHECKED(SQLGetData(

 hstmt,

 3,

 SQL\_C\_WCHAR,

 &rowData.status,

 sizeof(rowData.status),

 cbLen+1));

 CHECKED(SQLGetData(

 hstmt,

 5,

 SQL\_C\_WCHAR,

 &rowData.hostName,

 sizeof(rowData.hostName),

 cbLen+2));

#endif

 wprintf(L"%d\t%s\t%s\n", rowData.spid, rowData.status, rowData.hostName);

 }

 }

 catch(...){}

 if (hstmt)

 {

 SQLFreeHandle(SQL\_HANDLE\_STMT, hstmt);

 hstmt = NULL;

 }

 if (hdbc)

 {

 SQLDisconnect(hdbc);

 SQLFreeHandle(SQL\_HANDLE\_DBC, hdbc);

 hdbc = NULL;

 }

 if (henv)

 {

 SQLFreeHandle(SQL\_HANDLE\_ENV, henv);

 henv = NULL;

 }

}

**Appendix B: Sample OLE DB Application**

#include <stddef.h> // offsetof

#include <comdef.h> // wprintf

#include <sqloledb.h> // ISQLServerErrorInfo etc.

#include <msdasc.h> // IDataInitialize

#define SECURITY\_WIN32

#include <security.h> // GetUserNameEx

#include <atlbase.h> // CComPtr

#define CHECKED(\_hr) do {if(FAILED(hr=\_hr)){PrintHR(hr); throw;}}while(0)

#define CHECKED\_BOOLEAN(b) do {if(!b){ShowError(); throw;}}while(0)

#define PrintHR(hr) PrintHR\_internal(hr, \_\_FILE\_\_, \_\_LINE\_\_)

void PrintHR\_internal(HRESULT hr, char \* szFile, ULONG uLine)

{

 wprintf(L"Error HR = %d (0x%x) %hs(%d)\n", hr, hr, szFile, uLine);

}

void ShowError()

{

 // Retrieve the system error message for the last-error code

 LPVOID lpMsgBuf;

 DWORD dw = GetLastError();

 DWORD len=0;

 len=FormatMessageW(

 FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |

 FORMAT\_MESSAGE\_FROM\_SYSTEM |

 FORMAT\_MESSAGE\_IGNORE\_INSERTS,

 NULL,

 dw,

 MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),

 (LPWSTR) &lpMsgBuf,

 0, NULL );

 if (!len)

 {

 wprintf(L"Error from FormatMessage %d\n Original error %d\n", GetLastError(), dw);

 }

 else

 {

 wprintf(L"(%d) %s\n", dw, (WCHAR\*)lpMsgBuf);

 }

 LocalFree(lpMsgBuf);

}

int \_\_cdecl main()

{

 HRESULT hr;

 CComPtr<IDBInitialize> apIDBInit;

 CComPtr<IDataInitialize> apIDI;

 CComPtr<IDBCreateSession> apIDBCreateSession;

 CComPtr<IDBCreateCommand> apIDBCreateCommand;

 CComPtr<ICommandText> apCommand;

 CComPtr<ICommandPrepare> apICommandPrepare;

 CComPtr<IAccessor> apIAccessor;

 WCHAR wszUserName[MAX\_PATH];

 DWORD dwLength = MAX\_PATH;

 DBBINDSTATUS pDBBindStatus[1];

 DBBINDING paramBindings[] =

 {

 {

 1, // iOrdinal

 0, // obValue

 0, // obLength

 0, // obStatus

 NULL, // pTypeInfo

 NULL, // pObject

 NULL, // pBindExt

 DBPART\_VALUE, // dwPart

 DBMEMOWNER\_CLIENTOWNED, // dwMemOwner

 DBPARAMIO\_INPUT, // eParamIO

 MAX\_PATH, // cbMaxLen

 0, // dwFlags

 DBTYPE\_WSTR, // wType

 0, // bPrecision

 0, // bScale

 }

 };

 HACCESSOR hParamAccessor = NULL;

 try

 {

 CHECKED(CoInitializeEx(NULL, COINIT\_MULTITHREADED));

 CHECKED(CoCreateInstance(CLSID\_MSDAINITIALIZE, NULL, CLSCTX\_INPROC\_SERVER, IID\_IDataInitialize, (void\*\*)&apIDI));

 CHECKED(apIDI->GetDataSource(NULL, CLSCTX\_INPROC\_SERVER, L"Provider=SQLNCLI11; Integrated Security=SSPI; Data Source=.;", IID\_IDBInitialize, (IUnknown\*\*)&apIDBInit));

 CHECKED(apIDBInit->Initialize());

 CHECKED(apIDBInit->QueryInterface(IID\_IDBCreateSession, (void\*\*)&apIDBCreateSession));

 CHECKED(apIDBCreateSession->CreateSession(NULL, IID\_IDBCreateCommand, (IUnknown\*\*)&apIDBCreateCommand));

 CHECKED(apIDBCreateCommand->CreateCommand(NULL, IID\_ICommandText, (IUnknown\*\*)&apCommand));

 CHECKED(apCommand->SetCommandText(DBGUID\_DBSQL,

 L"{call sp\_who(?)}"));

 CHECKED(apCommand->QueryInterface(&apICommandPrepare));

 CHECKED(apICommandPrepare->Prepare(0));

 // Get the user name from the OS:

 CHECKED\_BOOLEAN(GetUserNameExW(NameSamCompatible, wszUserName, &dwLength));

 // Bind parameter

 CHECKED(apCommand->QueryInterface(&apIAccessor));

 CHECKED(apIAccessor->CreateAccessor(

 DBACCESSOR\_PARAMETERDATA,

 1,

 paramBindings,

 sizeof(wszUserName),

 &hParamAccessor,

 pDBBindStatus));

 {

 DBPARAMS dbParam =

 {

 wszUserName,

 1,

 hParamAccessor

 };

 struct sRowData

 {

 int spid;

 WCHAR status[30];

 WCHAR hostName[MAX\_PATH];

 } rowData;

 // We will bind only 3 columns: spid,status and hostname

 DBBINDING rgBinding[3] =

 // ordinal, obValue, obLength, obStatus, pTypeInfo, pObject, pBindExt,

 // dwPart, dwMemOwner, eParamIO, cbMaxLen, dwFlags, wType, bPrecision, bScale

 {

 {1, offsetof(sRowData, spid), 0, 0, 0, 0, 0,

 DBPART\_VALUE, DBMEMOWNER\_CLIENTOWNED, DBPARAMIO\_NOTPARAM, sizeof(rowData.spid), 0, DBTYPE\_I4, 0, 0},

 {3, offsetof(sRowData, status), 0, 0, 0, 0, 0,

 DBPART\_VALUE, DBMEMOWNER\_CLIENTOWNED, DBPARAMIO\_NOTPARAM, sizeof(rowData.status), 0, DBTYPE\_WSTR, 0, 0},

 {5, offsetof(sRowData, hostName), 0, 0, 0, 0, 0,

 DBPART\_VALUE, DBMEMOWNER\_CLIENTOWNED, DBPARAMIO\_NOTPARAM, sizeof(rowData.hostName), 0, DBTYPE\_WSTR, 0, 0},

 };

 HACCESSOR hRowAccessor = NULL;

 CComPtr<IRowset> apIRowset;

 CComPtr<IAccessor> apIRowAccessor;

 HROW rgRow[1];

 HROW \* rghRows = rgRow;

 CHECKED(apCommand->Execute(

 NULL, //pUnkOuter

 IID\_IRowset, //riid

 &dbParam,

 NULL, //pcRowsAffected

 (IUnknown\*\*)&apIRowset)); //ppRowset

 CHECKED(apIRowset->QueryInterface(&apIRowAccessor));

 CHECKED(apIRowAccessor->CreateAccessor(

 DBACCESSOR\_ROWDATA,

 3,

 rgBinding,

 sizeof(rowData),

 &hRowAccessor,

 pDBBindStatus));

 wprintf(L"spid\tstatus\t\t\t\thostname\n");

 while (hr == S\_OK)

 {

 DBCOUNTITEM cRows = 0;

 CHECKED(apIRowset->GetNextRows(

 DB\_NULL\_HCHAPTER,

 0,

 1,

 &cRows,

 &rghRows));

 if (cRows)

 {

 CHECKED(apIRowset->GetData(rgRow[0], hRowAccessor, &rowData));

 if (hr == S\_OK)

 {

 wprintf(L"%d\t%s\t%s\n", rowData.spid, rowData.status, rowData.hostName);

 }

 CHECKED(apIRowset->ReleaseRows(

 cRows,

 rghRows,

 NULL,

 NULL,

 NULL));

 }

 }

 }

 }

 catch(...){};

 CoUninitialize();

 return 0;

}