Programming ODBC SQL Server Applications

ODBC is a standard definition of an application programming interface (API) used to access data in relational or indexed sequential access method (ISAM) databases. Microsoft® SQL Server[™] supports ODBC as one of the native APIs for writing C, C++, and Microsoft Visual Basic® applications that communicate with SQL Server. SQL Server Setup installs an ODBC driver for use with SQL Server when it installs the SQL Server client utilities.

ODBC defines a call-level interface, or CLI. A CLI is defined as a set of function calls and their associated parameters. A CLI definition uses a native programming language to call functions; therefore a CLI requires no extensions to the underlying programming language. This contrasts with an embedded API, such as Embedded SQL, where the API is defined as an extension of the source code for a programming language, and applications using the API must be precompiled in a separate step.

ODBC aligns with the following specification and standard for relational SQL database CLI definitions:

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

While C, C++, and Visual Basic applications can be written to call ODBC directly, Microsoft also provides several APIs that map over ODBC. These APIs are simpler than ODBC itself, or offer improved integration with their respective programming languages:

- Microsoft Visual Basic Remote Data Objects (RDO)
- Microsoft Visual Basic Data Access Objects (DAO)
- Microsoft Visual C++® development system MFC database classes

• Microsoft Visual C++ development system DAO classes

While Visual Basic applications can be written directly to the ODBC API, they are usually written to either the RDO or DAO APIs.

SQL Server programs that are written using the ODBC API communicate with SQL Server through C function calls. The SQL Server-specific versions of the ODBC functions are implemented in a SQL Server ODBC driver. The driver passes SQL statements to SQL Server and returns the results of the statements to the application. ODBC applications are also interoperable with drivers for heterogeneous data sources.

The SQL Server ODBC driver complies with the Microsoft Win32® ODBC 3.51 specification. The ODBC function calls in this document use ODBC 3.51 syntax. The driver supports applications written with the ODBC 2.5 or earlier versions of the ODBC functions in the manner defined in the ODBC 3.51 specification.

ODBC enables a database to become an integral part of an application. SQL statements can be incorporated into the application, allowing the application to retrieve and update values from a database. Values from the database can be placed in program variables for manipulation by the application. Conversely, values in program variables can be written to the database.

ODBC enables applications to access a variety of data sources, including a wide range of relational databases and local ISAM data. ODBC supports applications in the Microsoft Windows® 2000, Microsoft Windows 98, Microsoft Windows 95, and Microsoft Windows NT® 4.0 operating environments.

Tools for developing C and C++ applications using the ODBC API are available in the Microsoft ODBC Software Development Kit (SDK). The ODBC SDK is part of the Microsoft Developer Network (MSDN®) Professional subscription. The ODBC SDK can also be downloaded from the <u>Microsoft Web site</u>, and is available in the *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference* available from Microsoft Press®. The ODBC driver for SQL Server is included with SQL Server. Visual Basic includes all the components necessary to build applications using the RDO and DAO APIs. Visual C++ includes all the components necessary to build C and C++ applications using the DAO and MFC database classes.

Getting Started with ODBC

These topics explain how to use ODBC to communicate with Microsoft® SQL ServerTM.

ODBC Syntax Conventions

Convention	Used for
UPPERCASE	Transact-SQL functions and statements, and C macro
	names.
monospace	Sample commands and program code.
italic	Function parameter names and information that the user or the application must provide.
bold	Function names, parameter keywords, and other syntax that must be typed exactly as shown.

System Requirements for ODBC

To access Microsoft® SQL Server[™] data, you must have the following software:

- SQL Server ODBC driver
- SQL Server
- Network software on the computers on which the driver and SQL Server reside (not required when connecting to a local (nonnetwork) desktop instance of SQL Server)

The hardware and software requirements of each of these components follow.

SQL Server ODBC Driver

The Microsoft[®] SQL Server[™] ODBC driver requires:

• Microsoft Windows® 2000, Microsoft Windows 95, or Microsoft Windows 98 on Intel computers.

Or

• Microsoft Windows NT® 4.0 on Intel computers.

For more information about the hardware and software required for SQL Server clients, see <u>Hardware and Software Requirements for Installing SQL Server</u>.

SQL Server

To use the Microsoft® SQL Server[™] ODBC driver to access data in SQL Server databases, you must have SQL Server version 4.21a or later. The catalog stored procedures must be installed on your SQL Server. You may need to install the catalog stored procedures shipped with this driver when you use versions 4.21a, 6.0, or 6.5 of SQL Server. For more information, see <u>Upgrading the Catalog</u> Stored Procedures (ODBC). For more information about the hardware and software required by ODBC SQL Server, see <u>Basic Installation Options</u>.

Network Software

Network software is required to connect the clients running the Microsoft® SQL Server[™] ODBC driver to the server on which the instance of SQL Server resides. To connect to a server running an instance of SQL Server, you can use Microsoft Windows NT® 4.0, Microsoft Windows® 2000, Microsoft Windows 95, Microsoft Windows 98, or a compatible network such as Novell NetWare or Banyan VINES. For more information about the hardware and software required by each network, see the network documentation.

The SQL Server ODBC driver communicates with network software through the SQL Server Net-Library interface, which requires a Net-Library dynamic-link library (DLL). The SQL Server 2000 ODBC driver requires the SQL Server 2000 versions of the Net-Library .dll files. These are installed when you run the client portion of SQL Server Setup. For more information about supported network configurations and Net-Library files, see <u>Basic Installation Options</u>.

Installing the SQL Server ODBC Driver

The Microsoft® SQL Server[™] ODBC driver is installed automatically when you install the SQL Server client software on a computer running Microsoft Windows NT® 4.0, Microsoft Windows® 2000, Microsoft Windows 98, or Microsoft Windows 95. For more information about installing SQL Server client software, see <u>Basic Installation Options</u>.

If you have servers running SQL Server versions 4.21a, 6.0, or 6.5, you must install the Instcat.sql file included with this driver on those servers before using the driver to access them. Each version of the SQL Server ODBC driver is developed in conjunction with a specific version of the catalog stored procedures. Instcat.sql upgrades the catalog stored procedures to the version required by the ODBC driver. This version of the catalog stored procedures is compatible with existing SQL Server applications.

Upgrading the Catalog Stored Procedures (ODBC)

The Microsoft® SQL Server[™] ODBC driver uses a set of system stored procedures, known as catalog stored procedures, to obtain information from the SQL Server system catalog. SQL Server installs the catalog stored procedures automatically when you install or upgrade SQL Server. The Instcat.sql file included with this driver includes minor updates to the catalog stored procedures. If this version of the SQL Server ODBC driver will be used against SQL Server version 6.5 or earlier versions, the SQL Server system administrator must upgrade the catalog stored procedures on the earlier SQL Server. Upgrading the catalog stored procedures does not affect the operation of existing SQL Server clients.

Running the SQL Server 2000 ODBC driver against an earlier version of SQL Server that has earlier versions of catalog stored procedures generates an error:

The ODBC catalog stored procedures installed on server <*server_nam* version <*old_version_number*>; version <*new_version_number*> or la required to ensure proper operation. Please contact your system administrator.

To upgrade the catalog stored procedures

Adding a Data Source

ODBC applications typically connect to a database through an ODBC data source. Each ODBC data source on a client computer has a unique data source name, or DSN. An ODBC data source for the Microsoft® SQL Server[™] ODBC driver includes all the information required to connect to a server running an instance of SQL Server, plus options, such as a default database or the type of security to use.

There are three types of ODBC data sources:

• User data source

User data sources are specific to the Microsoft Windows NT® 4.0, Microsoft Windows® 2000, Microsoft Windows 95, or Microsoft Windows 98 account in effect when they are created. They are not visible to any other login account. They are not always visible to applications running as a service on a Windows NT 4.0 computer.

• System data source

System data sources are visible to all login accounts on a client. They are always visible to applications running as a service on a Windows NT 4.0 computer.

• File data source

File data sources were added with ODBC version 3.0. File data sources are not stored in the system registry. They are stored in a file on the client.

There are several ways to add a data source:

• ODBC Administrator

The ODBC Administrator is installed in Control Panel. The ODBC Administrator has tabs for user, system, and file data sources. Click the proper tab, click **Add**, and then select the SQL Server ODBC driver. The ODBC Administrator then starts the SQL Server DSN Configuration Wizard.

• SQLConfigDataSource

User or system data sources can be created by an ODBC application that calls the **SQLConfigDataSource** function with the *fRequest* parameter set to either ODBC_ADD_DSN or ODBC_ADD_SYS_DSN.

• SQLWriteFileDSN

A file data source can be created by an ODBC application that calls the **SQLWriteFileDSN** function.

• SQLDriverConnect

If an application specifies the SAVEFILE keyword in the connect string of a successful call to **SQLDriverConnect**, a file data source is created using the information specified in the **SQLDriverConnect** connect string.

• SQLCreateDataSource

An ODBC application can call the function **SQLCreateDataSource** to display an ODBC dialog box that guides a user through creating a data source.

Data sources that reference the SQL Server ODBC driver contain driver-specific information and options. When a data source is created with either **SQLConfigDataSource** or **SQLWriteFileDSN**, all of the driver-specific information is supplied through keyword-value pairs in a character string passed to the function. When a data source is created using the ODBC Administrator or the **SQLCreateDataSource** dialog box, the SQL Server DSN Creation wizard is invoked to help you perform the steps to specify the driver-specific information.

The help file for the SQL Server DSN Creation Wizard contains information on the driver-specific options defined through the wizard. These options can also be specified as keywords in **SQLConfigDataSource**. For more information about the driver-specific options that can be specified with **SQLConfigDataSource**, see **SQLConfigDataSource**. The same driver-specific options can also be specified as keywords in the connect string for **SQLDriverConnect**. For more information about the keywords and their meanings, see <u>SQLDriverConnect</u>.

To start the Microsoft SQL Server DSN Configuration Wizard

Deleting a Data Source

ODBC data sources can be deleted in several ways:

- Double-click the **ODBC Administrator** icon in Control Panel, select the data source, and then click **Delete**.
- Call **SQLConfigDataSource** with the *fRequest* parameter set to either SQL_REMOVE_DSN or SQL_REMOVE_SYS_DSN.
- Delete file data sources by deleting the file containing the data source.

To delete a data source

Connecting to a SQL Server Data Source

After an ODBC data source has been defined, you can connect to an instance of Microsoft® SQL ServerTM from ODBC applications using the data source. Some ODBC applications are written to connect through data sources and typically open a dialog box or accept a parameter for the ODBC data source with which you want to connect. Other ODBC applications are written to connect without a data source. These applications can display their own dialog box to get the information required to connect, or they can have the ODBC driver display its dialog box to get the connection information. SQL Query Analyzer is an example of an ODBC application that does not use a data source; Microsoft Excel is an example of an ODBC application that does use a data source.

Using odbcping to Verify a Connection

You can use the **odbcping** utility to check whether ODBC is properly installed by connecting to a server using the Microsoft® SQL Server[™] ODBC driver. This utility is a 32-bit application stored in the \Mssql7\Binn directory.

To verify ODBC connectivity

∃ <u>ODBC</u>

Creating an ODBC Application

ODBC architecture has four components that perform the following functions.

Component	Function
Application	Calls ODBC functions to communicate with an ODBC
	data source, submits SQL statements, and processes
	result sets.
Driver Manager	Manages communication between an application and
	all ODBC drivers used by the application.
Driver	Processes all ODBC function calls from the
	application, connects to a data source, passes SQL
	statements from the application to the data source, and
	returns results to the application. If necessary, the
	driver translates ODBC SQL from the application to
	native SQL used by the data source.
Data source	Contains all information a driver needs to access a
	specific instance of data in a DBMS.

An application that uses the ODBC interface to communicate with an instance of Microsoft® SQL Server[™] performs the following tasks:

- Connects with a data source
- Sends SQL statements to the data source
- Processes the results of statements from the data source
- Processes errors and messages
- Terminates the connection to the data source

A more complex application written for the SQL Server ODBC driver might also perform the following tasks:

- Use cursors to control location in a result set
- Request commit or rollback operations for transaction control
- Perform distributed transactions involving two or more servers
- Run stored procedures on the remote server
- Call catalog functions to inquire about the attributes of a result set
- Perform bulk copy operations
- Manage long data (text, ntext, and image columns) operations
- Control failover servers in case the primary server becomes unavailable
- Log performance data and long-running queries

To make ODBC function calls, a C or C++ application must include the Sql.h, Sqlext.h, and Sqltypes.h header files. To make calls to the ODBC installer API functions, an application must include the Odbcinst.h header file. A Unicode ODBC application must include the Sqlucode.h header file. ODBC applications must be linked with the Odbc32.lib file. ODBC applications that call the ODBC installer API functions must be linked with the Odbccp32.lib file. By default, SQL Server Setup 2000 installs these header files into the C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Include directory and the library files into C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Lib when the SQL Server development tools are installed. The latest versions of these files can be downloaded with the latest Microsoft Data Access SDK from the Microsoft Web site.

Many ODBC drivers, including the SQL Server ODBC driver, offer driverspecific ODBC extensions. To take advantage of SQL Server ODBC driverspecific extensions, an application should include the Odbcss.h header file. This header file contains:

- SQL Server ODBC driver-specific connection attributes.
- SQL Server ODBC driver-specific statement attributes.
- SQL Server ODBC driver-specific column attributes.
- SQL Server-specific data types.
- SQL Server-specific user-defined data types.
- SQL Server ODBC driver-specific **SQLGetInfo** types.
- SQL Server ODBC driver diagnostics fields.
- SQL Server-specific diagnostic dynamic function codes.
- C/C++ type definitions for SQL Server-specific native C data types (returned when columns bound to C data type SQL_C_BINARY).
- Type definition for the SQLPERF data structure.
- Bulk copy macros and prototypes to support bulk copy API usage through an ODBC connection.
- Call the distributed query meta data API functions for lists of linked servers and their catalogs.

Any C or C++ ODBC application that uses the bulk copy feature of the SQL

Server 2000 ODBC driver must be linked with the Odbcbcp.lib file. Applications calling the distributed query meta data API functions must also be linked with Odbcbcp.lib. The Odbcss.h and Odbcbcp.lib files are distributed as part of the SQL Server developer's tools. The SQL Server Include and Lib directories should be in the compiler's INCLUDE and LIB paths. If you have downloaded a version of the Microsoft Data Access SDK whose dates are later than the dates for SQL Server version 7.0, place the MSDA directories before the SQL Server 7.0 directories; for example:

LIB=c:\msdasdk\odbc\lib;C:\Program Files\Microsoft SQL Server\80\ INCLUDE=c:\msdasdk\odbc\include;C:\Program Files\Microsoft SQL c:\msdev\mfc\include

One design decision made early in the process of building an application is whether the application needs to have multiple ODBC calls outstanding at the same time. There are two methods for supporting multiple concurrent ODBC calls:

- ODBC asynchronous mode
- Multithreading

Asynchronous Mode and SQLCancel

Some ODBC functions can operate either synchronously or asynchronously. (For more information about ODBC functions, see the *ODBC 3.0 Programmer's Reference*.) The application can enable asynchronous operations for either a statement handle or a connection handle. If the option is set for a connection handle, it affects all statement handles on the connection handle. The application uses the following statements to enable or disable asynchronous operations:

SQLSetConnectAttr(hdbc, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, SQL_NTS); SQLSetConnectAttr(hdbc, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_OFF, SQL_NTS); SQLSetStmtAttr(hstmt, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, SQL_NTS); SQLSetStmtAttr(hstmt, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_OFF, SQL_NTS);

When an application calls an ODBC function in synchronous mode, the driver does not return control to the application until it is notified that the server has completed the command.

When operating asynchronously, the driver immediately returns control to the application, even before sending the command to the server. The driver sets the return code to SQL_STILL_EXECUTING. The application can then perform other work.

When the application tests for completion of the command, it makes the same function call with the same parameters to the driver. If the driver has not yet received an answer from the server, it will again return SQL_STILL_EXECUTING. The application must test the command periodically until the return code is something other than SQL_STILL_EXECUTING. When the application gets some other return code, even SQL_ERROR, it knows the command has completed.

Sometimes a command is outstanding for a long time. If the application needs to cancel the command without waiting for a reply, it can do so by calling

SQLCancel with the same statement handle as the outstanding command. This is the only time **SQLCancel** should be used. Some programmers use **SQLCancel** when they have processed part way through a result set and want to cancel the rest of the result set. **SQLMoreResults** or **SQLCloseCursor** should be used to cancel the remainder of an outstanding result set, not **SQLCancel**.

See Also

SQLCloseCursor

SQLMoreResults

Multithreaded Applications

The Microsoft® SQL Server[™] ODBC driver is a multithreaded driver. Writing a multithreaded application is an alternative to using asynchronous calls to process multiple ODBC calls. A thread can make a synchronous ODBC call, and other threads can process while the first thread is blocked waiting for the response to its call. This model is more efficient than making asynchronous calls because it eliminates overhead such as network traffic and making repeated ODBC function calls testing for SQL_STILL_EXECUTING.

Asynchronous mode is still an effective method of processing. The performance improvements of a multithreaded model are not enough to justify rewriting asynchronous applications. If users are converting DB-Library applications that use the DB-Library asynchronous model, it is easier to convert them to the ODBC asynchronous model.

Communicating with SQL Server

For an ODBC application to communicate with an instance of Microsoft® SQL Server[™], it must allocate environment and connection handles and connect to the data source. After a connection is established, the application can send queries to the server and process any result sets. When the application has finished using the data source, it disconnects from the data source and frees the connection handle. When the application has freed all of its connection handles, it frees the environment handle.

An application can connect to any number of data sources. The application can use a combination of drivers and data sources, the same driver and a combination of data sources, or even the same driver and multiple connections to the same data source.

See Also

<u>SQLSetEnvAttr</u>

Allocating an Environment Handle

Before an application can call any ODBC function, it must initialize the ODBC environment and allocate an environment handle, which is the global context handle and placeholder for the other handles in ODBC. This is done by calling **SQLAllocHandle** with the *HandleType* parameter set to SQL_HANDLE_ENV and *InputHandle* set to SQL_NULL_HANDLE.

After allocating the environment handle, the application must set environment attributes to indicate which version of ODBC function calls it will be using. To use the ODBC 3.*x* functions, call **SQLSetEnvAttr** with the *Attribute* parameter set to SQL_ATTR_ODBC_VERSION and *ValuePtr* set to SQL_OV_ODBC3.

How to allocate handles and connect to SQL Server

Allocating a Connection Handle

Before the application can connect to a data source or driver, it must allocate a connection handle. This is done by calling **SQLAllocHandle** with the *HandleType* parameter set to SQL_HANDLE_DBC and *InputHandle* pointing to an initialized environment handle.

The characteristics of the connection are controlled by setting connection attributes. For example, because transactions occur at the connection level, the transaction isolation level is a connection attribute. Similarly, the login time-out, or number of seconds to wait while trying to connect before timing out, is a connection attribute.

Connection attributes are set with **SQLSetConnectAttr**, and their current settings are retrieved with **SQLGetConnectAttr**. If **SQLSetConnectAttr** is called before a connection is attempted, the ODBC Driver Manager stores the attributes in its connection structure and sets them in the driver as part of the connection process. Some connection attributes must be set before the application attempts to connect; others can be set after the connection has completed. For example, SQL_ATTR_ODBC_CURSORS must be set before a connection is made, but SQL_ATTR_AUTOCOMMIT can be set after connecting.

Applications running against Microsoft® SQL Server[™] version 6.0 or later can sometimes improve their performance by resetting the Tabular Data Stream (TDS) network packet size. The default packet size is set at the server, at 4 KB. A packet size of 4 KB to 8 KB generally gives the best performance. If testing shows that it performs better with a different packet size, the application can reset the packet size. ODBC applications can do this before connecting by calling **SQLSetConnectionAttr** with the SQL_ATTR_PACKET_SIZE option. Some applications perform better with a larger packet size, but performance improvements are generally minimal for packet sizes larger than 8 KB.

The SQL Server ODBC driver has a number of extended connection attributes that an application can use to increase its functionality. Some of these attributes control the same options that can be specified in data sources and used to override whatever option is set in a data source. For example, if an application uses quoted identifiers, it can set the driver-specific attribute SQL_COPT_SS_QUOTED_IDENT to SQL_QI_ON to ensure this option is always set regardless of the setting in any data source.

How to allocate handles and connect to SQL Server

SQL Server ODBC Data Sources

A Microsoft[®] SQL Server[™] data source name (DSN) identifies an ODBC data source containing all of the information that an ODBC application needs to connect to a SQL Server database on a specific server. There are two ways you can define an ODBC data source name:

- On a client computer, in Control Panel, double-click 32-bit ODBC.
- In an ODBC application, call <u>SQLConfigDataSource</u>.

A SQL Server data source contains:

- The name of the data source.
- Any information needed to connect to a specific instance of SQL Server.
- The default database to use on a specific instance of SQL Server (optional).
- Settings such as which ANSI options to use, whether to log performance statistics, and so on (optional).

An ODBC application is not required to connect through a data source. However, the application must provide the same connectivity information to an ODBC connect function that the driver would otherwise find in a DSN.

Connecting to a Data Source

After allocating environment and connection handles and setting any connection attributes, the application connects to the data source or driver. There are three functions you can use to connect:

- SQLConnect
- SQLDriverConnect
- SQLBrowseConnect

SQLConnect

SQLConnect is the simplest connection function. It accepts three parameters: a data source name, a user ID, and a password. Use **SQLConnect** when these three parameters contain all the information needed to connect to the database. To do this, build a list of data sources using **SQLDataSources**; prompt the user for a data source, user ID, and password; and then call **SQLConnect**.

SQLConnect assumes that a data source name, user ID, and password are sufficient to connect to a data source and that the ODBC data source contains all other information the ODBC driver needs to make the connection. Unlike **SQLDriverConnect** and **SQLBrowseConnect**, **SQLConnect** does not use a connection string.

SQLDriverConnect

SQLDriverConnect is used when more information than the data source name, user ID, and password is required. One of the parameters to **SQLDriverConnect** is a connection string containing driver-specific information. You might use **SQLDriverConnect** instead of **SQLConnect** for the following reasons:

- To specify driver-specific information at connect time.
- To request that the driver prompt the user for connection information.

• To connect without using an ODBC data source.

The **SQLDriverConnect** connection string contains a series of keyword-value pairs that specify all connection information supported by an ODBC driver. Each driver supports the standard ODBC keywords (DSN, FILEDSN, DRIVER, UID, PWD, and SAVEFILE) in addition to driver-specific keywords for all connection information supported by the driver. **SQLDriverConnect** can be used to connect without a data source. For example, an application that is designed to make a "DSN-less" connection to an instance of Microsoft® SQL Server[™] can call **SQLDriverConnect** with a connection string that defines the login ID, password, network library, server name to connect to, and default database to use.

When using **SQLDriverConnect**, there are two options for prompting the user for any needed connection information:

• Application dialog box

You can create an application dialog box that prompts for connection information, and then calls **SQLDriverConnect** with a NULL window handle and *DriverCompletion* set to SQL_DRIVER_NOPROMPT. These parameter settings prevent the ODBC driver from opening its own dialog box. This method is used when it is important to control the user interface of the application.

• Driver dialog box

You can code the application to pass a valid window handle to **SQLDriverConnect** and set the *DriverCompletion* parameter to SQL_DRIVER_COMPLETE, SQL_DRIVER_PROMPT, or SQL_DRIVER_COMPLETE_REQUIRED. The driver will then generate a dialog box to prompt the user for connection information. This method simplifies the application code.

SQLBrowseConnect

SQLBrowseConnect, like **SQLDriverConnect**, uses a connection string. However, by using **SQLBrowseConnect**, an application can construct a complete connection string iteratively with the data source at run time. This allows the application to do two things:

- Build its own dialog boxes to prompt for this information, thereby retaining control over its user interface.
- Browse the system for data sources that can be used by a particular driver, possibly in several steps.

For example, the user might first browse the network for servers and, after choosing a server, browse the server for databases accessible by the driver.

When **SQLBrowseConnect** completes a successful connection, it returns a connection string that can be used on subsequent calls to **SQLDriverConnect**.

The SQL Server ODBC driver always returns SQL_SUCCESS_WITH_INFO on a successful **SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**. When an ODBC application calls **SQLGetDiagRec** after getting SQL_SUCCESS_WITH_INFO, it can receive the following messages:

5701

Indicates that SQL Server put the user's context into the default database defined in the data source, or into the default database defined for the login ID used in the connection if the data source did not have a default database.

5703

Indicates the language being used on the server.

The following example shows the message returned on a successful connection by the system administrator:

```
szSqlState = "01000", *pfNativeError = 5701,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
Changed database context to 'pubs'."
szSqlState = "01000", *pfNativeError = 5703,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
Changed language setting to 'us_english'."
```

You can ignore messages 5701 and 5703; they are only informational. You

should not, however, ignore a SQL_SUCCESS_WITH_INFO return code because messages other than 5701 or 5703 may be returned. For example, if a driver connects to a server running an instance of SQL Server with outdated catalog stored procedures, one of the errors returned through **SQLGetDiagRec** after a SQL_SUCCESS_WITH_INFO is:

SqlState: 01000 pfNative: 0 szErrorMsg: "[Microsoft][ODBC SQL Server Driver]The ODBC catalog stored procedures installed on server my65server are version 06.50.0193; version 07.00.0205 or later is required to ensure proper operation. Please contact your system administrator."

The error handling function of an application for SQL Server connections should call **SQLGetDiagRec** until it returns SQL_NO_DATA. It should then act on any messages other than the ones with a *pfNative* code of 5701 or 5703.

To use connections

Disconnecting from a Data Source

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. After disconnecting, the application can call **SQLFreeHandle** to free the connection handle. Before exiting, an application also calls **SQLFreeHandle** to free the environment handle.

After disconnecting, an application can reuse the allocated connection handle, either to connect to a different data source, or to reconnect to the same data source. The decision to remain connected, as opposed to disconnecting and reconnecting later, requires that the application writer consider the relative costs of each option: both connecting to a data source and remaining connected can be relatively costly, depending on the connection medium. In making a correct tradeoff, the application must also make assumptions about the likelihood and timing of further operations on the same data source. An application may also need to use more than one connection.

To use connections

Executing Queries

After an ODBC application initializes a connection handle and connects with a data source, it allocates one or more statement handles on the connection handle. The application can then execute Microsoft® SQL Server[™] statements on the statement handle. The general sequence of events in executing an SQL statement is:

- 1. Set any required statement attributes.
- 2. Construct the statement.
- 3. Execute the statement.
- 4. Retrieve any result sets.

After an application retrieves all of the rows in all of the result sets returned by the SQL statement, it can execute another query on the same statement handle. If an application determines that it is not required to retrieve all of the rows in a particular result set, it can cancel the remainder of the result set by calling either **SQLMoreResults** or **SQLCloseCursor**.

If, in an ODBC application, it is necessary to execute the same SQL statement multiple times with different data, use a parameter marker, denoted by a question mark (?), in the construction of an SQL statement:

INSERT INTO MyTable VALUES (?, ?, ?)

Each parameter marker can then be bound to a program variable by calling **SQLBindParameter**.

After all SQL statements execute and their result sets process, the application frees the statement handle.

The SQL Server ODBC driver supports multiple statement handles per connection handle. Transactions are managed at the connection level, so all work

done on all statement handles on a single connection handle are managed as part of the same transaction.

See Also

SQLBindParameter SQLCloseCursor SQLMoreResults

Allocating a Statement Handle

Before an application can execute a statement, it must allocate a statement handle. It does this by calling **SQLAllocHandle** with the *HandleType* parameter set to SQL_HANDLE_STMT and *InputHandle* pointing to a connection handle.

Statement attributes are characteristics of the statement handle. Sample statement attributes can include whether to use bookmarks and what kind of cursor to use with the statement's result set. Statement attributes are set with **SQLSetStmtAttr**, and their current settings are retrieved with **SQLGetStmtAttr**. There is no requirement that an application set any statement attributes; all statement attributes have defaults; some are driver-specific.

Use caution in the use of several ODBC statement and connection options. Calling **SQLSetConnectAttr** with *fOption* set to

SQL_ATTR_LOGIN_TIMEOUT controls the amount of time an application waits for a connection attempt to timeout while waiting to establish a connection (0 specifies an infinite wait). Sites with slow response times can set this value high to ensure connections have sufficient time to complete, but the interval should always be low enough to give the user a response in a reasonable amount of time if the driver cannot connect.

Calling **SQLSetStmtAttr** with *fOption* set to SQL_ATTR_QUERY_TIMEOUT sets a query time-out interval to protect the server and the user from long-running queries.

Calling **SQLSetStmtAttr** with *fOption* set to SQL_ATTR_MAX_LENGTH limits the amount of **text** and **image** data that an individual statement can retrieve. Calling **SQLSetStmtAttr** with *fOption* set to SQL_ATTR_MAX_ROWS also limits a rowset to the first *n* rows if that is all the application requires. Note that setting SQL_ATTR_MAX_ROWS causes the driver to issue a SET ROWCOUNT statement to the server, which affects all Microsoft® SQL Server[™] statements, including triggers and updates.

Use caution when setting these options. It is best if all statement handles on a connection handle have the same settings for SQL_ATTR_MAX_LENGTH and SQL_ATTR_MAX_ROWS. If the driver switches from a statement handle to another with different values for these options, the driver must generate the

appropriate SET TEXTSIZE and SET ROWCOUNT statements to change the settings. The driver cannot put these statements in the same batch as the user SQL statement because the user SQL statement can contain a statement that must be the first statement in a batch. The driver must send the SET TEXTSIZE and SET ROWCOUNT statements in a separate batch, which automatically generates an extra roundtrip to the server.

To use a statement

Constructing an SQL Statement

ODBC applications perform almost all of their database access by executing Microsoft® SQL Server[™] statements. The form of these statements depends on the application requirements. SQL statements can be constructed in the following ways:

• Hard-coded

Static statements performed by an application as a fixed task.

• Constructed at run time

SQL statements constructed at run time that enable the user to tailor the statement by using common clauses, such as SELECT, WHERE, and ORDER BY. This includes ad hoc queries entered by users.

The SQL Server ODBC driver parses SQL statements only for ODBC and SQL-92 syntax not directly supported by the database engine, which the driver transforms into Transact-SQL. All other SQL syntax is passed to the database engine unchanged, where SQL Server will determine if it is valid Transact-SQL. This approach yields two benefits:

• Reduced overhead

Processing overhead for the driver is minimized because it only has to scan for a small set of ODBC and SQL-92 clauses.

• Flexibility

Programmers can tailor the portability of their applications. To enhance portability against multiple databases, use primarily ODBC and SQL-92 syntax. To use enhancements specific to SQL Server, use the appropriate Transact-SQL syntax. The SQL Server ODBC driver supports the complete Transact-SQL syntax so ODBC-based applications can take advantage of all the features in SQL Server.

The column list in a SELECT statement should contain only the columns required to perform the current task. Not only does this reduce the amount of data sent across the network, but also it reduces the effect of database changes on the application. If an application does not reference a column from a table, then the application is not affected by any changes made to that column.

To use statements

Constructing SQL Statements for Cursors

The Microsoft® SQL Server[™] ODBC driver uses server cursors to implement the cursor functionality defined in the ODBC specification. An ODBC application controls the cursor behavior by using **SQLSetStmtAttr** to set different statement attributes. These are the attributes and their defaults.

Attribute	Default
SQL_ATTR_CONCURRENCY	SQL_CONCUR_READ_ONLY
SQL_ATTR_CURSOR_TYPE	SQL_CURSOR_FORWARD_ONLY
SQL_ATTR_CURSOR_SCROLLABLE	SQL_NONSCROLLABLE
SQL_ATTR_CURSOR_SENSITIVITY	SQL_UNSPECIFIED
SQL_ATTR_ROW_ARRAY_SIZE	1

When these options are set to their defaults at the time an SQL statement is executed, the SQL Server ODBC driver does not use a server cursor to implement the result set; instead, it uses a default result set. If any of these options are changed from their defaults at the time an SQL statement is executed, the SQL Server ODBC driver attempts to use a server cursor to implement the result set.

Default result sets support all of the Transact-SQL statements. There are no restrictions on the types of SQL statements that can be executed when using a default result set.

Server cursors do not support all Transact-SQL statements. Server cursors do not support any SQL statement that generates multiple result sets.

The following types of statements are not supported by server cursors:

• Batches

SQL statements built from two or more individual SQL SELECT statements, for example:

SELECT * FROM authors; SELECT * FROM titles

• Stored procedures with multiple SELECT statements

SQL statements that execute a stored procedure containing more than one SELECT statement. This includes SELECT statements that fill parameters or variables.

• Keywords

SQL statements containing the keywords COMPUTE, COMPUTE BY, FOR BROWSE, or INTO.

In SQL Server, if an SQL statement that matches any of these conditions is executed with a server cursor, the server cursor is implicitly converted to a default result set. After **SQLExecDirect** or **SQLExecute** returns SQL_SUCCESS_WITH_INFO, the cursor attributes will be set back to their default settings.

In SQL Server version 6.5 or earlier, these statements cannot be executed with any of the statement attribute settings that would generate a server cursor. **SQLExecDirect** or **SQLExecute** return SQL_ERROR unless the cursor attributes are first set to their defaults to generate a default result set.

SQL statements that do not fit the categories above can be executed with any statement attribute settings; they work equally well with either a default result set or a server cursor.

Errors

In SQL Server 7.0, an attempt to execute a statement that produces multiple result sets generates SQL_SUCCESS_WITH INFO and the following message:

```
SqlState: 01S02"
pfNative: 0
szErrorMsgString: "[Microsoft][ODBC SQL Server Driver]
Cursor type changed."
```

ODBC applications receiving this message can call **SQLGetStmtAttr** to determine the current cursor settings.

Attempting to execute statements that generate multiple results in SQL Server version 6.5 or earlier generates SQL_ERROR and one of the following messages depending on the type of statement executed.

An attempt to execute a procedure with multiple SELECT statements when using server cursors generates the following error:

SqlState: 42000 pfNative: 16937 szErrorMsgString: [Microsoft][ODBC SQL Server Driver][SQL Serve A server cursor is not allowed on a stored procedure with more than one SELECT statement in it. Use a default result set or client cursor.

An attempt to execute a batch with multiple SELECT statements when using server cursors generates the following error:

SqlState: 42000 pfNative: 16938 szErrorMsgString: [Microsoft][ODBC SQL Server Driver][SQL Serve sp_cursoropen. The statement parameter can only be a single SELECT statement or a single stored procedure.

An attempt to execute a SELECT statement containing a COMPUTE clause when using server cursors generates the following error:

SqlState: 42000 pfNative: 16907 szErrorMsgString: [Microsoft][ODBC SQL Server Driver][SQL Serve 'COMPUTE' is not allowed in cursor statements.

ODBC applications receiving these errors must reset all the cursor statement attributes to their defaults before attempting to execute the statement.

To set cursor options

Using Statement Parameters

A parameter is a variable in an SQL statement that can enable an ODBC application to:

- Efficiently provide values for columns in a table.
- Enhance user interaction in constructing query criteria.
- Manage **text**, **ntext**, and **image** data and Microsoft® SQL Server[™]-specific C data types.

For example, a **parts** table has columns named **partid**, **description**, and **price**. To add a part without parameters requires constructing an SQL statement such as:

INSERT INTO Parts (PartID, Description, Price) VALUES (2100, 'Dri

Although this statement is acceptable for inserting one row with a known set of values, it is awkward if an application is required to insert several rows. ODBC addresses this by allowing an application to replace any data value in an SQL statement by a parameter maker, which is denoted by a question mark (?). In the following example, three data values are replaced with parameter markers:

INSERT INTO Parts (PartID, Description, Price) VALUES (?, ?, ?)

The parameter markers are then bound to application variables. To insert a new row, the application has only to set the values of the variables and execute the statement. The driver then retrieves the current values of the variables and sends them to the data source. If the statement will be executed multiple times, the application can make the process even more efficient by preparing the statement.

Each parameter marker is referenced by its ordinal number, assigned to the parameters from left to right. The leftmost parameter marker in an SQL statement has an ordinal value of 1, the next one is ordinal 2, and so on.

To execute a statement directly

Binding Parameters

Each parameter marker in an SQL statement must be associated, or bound, to a variable in the application before the statement can be executed. This is done by calling the **SQLBindParameter** function. **SQLBindParameter** describes the program variable (address, C data type, and so on) to the driver. It also identifies the parameter marker by indicating its ordinal value and then describes the characteristics of the SQL object it represents (SQL data type, precision, and so on).

Parameter markers can be bound or rebound at any time before a statement is executed. A parameter binding remains in effect until one of the following occurs:

- A call to **SQLFreeStmt** with the *Option* parameter set to SQL_RESET_PARAMS frees all parameters bound to the statement handle.
- A call to **SQLBindParameter** with *ParameterNumber* set to the ordinal of a bound parameter marker automatically releases the previous binding.

An application can also bind parameters to arrays of program variables to process an SQL statement in batches. There are two types of array binding:

• Column-wise binding is done when each individual parameter is bound to its own array of variables.

Column-wise binding is specified by calling **SQLSetStmtAttr** with *Attribute* set to SQL_ATTR_PARAM_BIND_TYPE and *ValuePtr* set to SQL_PARAM_BIND_BY_COLUMN.

• Row-wise binding is done when all of the parameters in the SQL statement are bound as a unit to an array of structures that contain the individual variables for the parameters.

Row-wise binding is specified by calling **SQLSetStmtAttr** with

Attribute set to SQL_ATTR_PARAM_BIND_TYPE and *ValuePtr* set to the size of the structure holding the program variables.

When the Microsoft® SQL Server[™] ODBC driver sends character or binary string parameters to the server, it pads the values to the length specified in **SQLBindParameter** *ColumnSize* parameter. If an ODBC 2.x application specifies 0 for *ColumnSize*, the driver pads the parameter value to the precision of the data type. The precision is 8000 when connected to SQL Server servers, 255 when connected to earlier versions of SQL Server. *ColumnSize* is in bytes for variant columns.

SQL Server supports defining names for stored procedure parameters. ODBC 3.5 also introduced support for named parameters used when calling SQL Server stored procedures. This support can be used to:

- Call a stored procedure and provide values for a subset of the parameters defined for the stored procedure.
- Specify the parameters in a different order in the application than the order specified when the stored procedure was created.

Named parameters are only supported when using the Transact-SQL EXECUTE statement or the ODBC CALL escape sequence to execute a stored procedure.

For more information about examples of using named parameters, see *ODBC 3.0 Software Developers Kit and Programmer's Reference*.

See Also

SQLBindParameter SQLFreeStmt SQLSetStmtAttr

Executing Statements

The ODBC API offers two ways to execute a Microsoft® SQL Server[™] statement:

- Direct execution
- Prepared execution

These two methods can execute a single SQL statement, a call of a stored procedure, or a batch of SQL statements.

Direct Execution

Direct execution is the most basic way to execute a statement. An application builds a character string containing a Microsoft® SQL Server[™] statement and submits it for execution using the **SQLExecDirect** function. When the statement reaches the server, SQL Server compiles it into an execution plan and then immediately runs the execution plan.

Direct execution is commonly used by applications that build and execute statements at run time and is the most efficient method for statements that will be executed a single time. Its drawback with many databases is that the SQL statement must be parsed and compiled each time it is executed, which adds overhead if the statement is executed multiple times.

When connected to versions of SQL Server earlier than 7.0, direct execution should be used:

- When a statement is likely to be executed fewer than four times.
- To call stored procedures.

SQL Server 2000 significantly improves the performance of direct execution of commonly executed statements in multiuser environments. For SQL Server 7.0 applications, using **SQLExecDirect** with parameter markers for commonly executed SQL statements can approach the efficiency of prepared execution.

When connected to an instance of SQL Server 2000, the SQL Server ODBC driver uses **sp_executesql** to transmit the SQL statement or batch specified on **SQLExecDirect**. SQL Server 2000 has logic to quickly determine if an SQL statement or batch executed with **sp_executesql** matches the statement or batch that generated an execution plan that already exists in memory. If a match is made, SQL Server simply reuses the existing plan rather than compile a new plan. This means that commonly executed SQL statements executed with **SQLExecDirect** in a system with many users will benefit from many of the plan-reuse benefits that were only available to stored procedures in earlier versions of SQL Server.

This benefit of reusing execution plans only works when several users are executing the same SQL statement or batch. Follow these coding conventions to increase the probability that the SQL statements executed by different clients are similar enough to be able to reuse execution plans:

- Do not include data constants in the SQL statements; instead use parameter markers bound to program variables. For more information, see <u>Using Statement Parameters</u>.
- Use fully qualified object names. Execution plans are not reused if object names are not qualified.
- Have application connections as possible use a common set of connection and statement options. Execution plans generated for a connection with one set of options (such as ANSI_NULLS) are not reused for a connection having another set of options. The SQL Server ODBC driver and the OLE DB Provider for SQL Server both have the same default settings for these options.

If all statements executed with **SQLExecDirect** are coded using these conventions, SQL Server can reuse execution plans when the opportunity arises.

To use a statement

Prepared Execution

The ODBC API defines prepared execution as a way to reduce the parsing and compiling overhead associated with repeatedly executing a Microsoft® SQL Server[™] statement. The application builds a character string containing an SQL statement and then executes it in two stages. It calls **SQLPrepare** once to have the statement parsed and compiled into an execution plan by the database engine. It then calls **SQLExecute** for each execution of the prepared execution plan. This saves the parsing and compiling overhead on each execution. Prepared execution is commonly used by applications to repeatedly execute the same, parameterized SQL statement.

For most databases, prepared execution is faster than direct execution for statements executed more than three or four times primarily because the statement is compiled only once, while statements executed directly are compiled each time they are executed. Prepared execution can also provide a reduction in network traffic because the driver can send an execution plan identifier and the parameter values, rather than an entire SQL statement, to the data source each time the statement is executed.

SQL Server 2000 reduces the performance difference between direct and prepared execution through improved algorithms for detecting and reusing execution plans from **SQLExecDirect**. This makes some of the performance benefits of prepared execution available to statements executed directly. For more information, see <u>Direct Execution</u>.

SQL Server 2000 also provides native support for prepared execution. An execution plan is built on **SQLPrepare** and later executed when **SQLExecute** is called. Because SQL Server 2000 is not required to build temporary stored procedures on **SQLPrepare**, there is no extra overhead on the system tables in **tempdb**.

For performance reasons, the statement preparation is deferred until **SQLExecute** is called or a metaproperty operation (such as **SQLDescribeCol** or **SQLDescribeParam** in ODBC) is performed. This is the default behavior. Any errors in the statement being prepared are not known until the statement is executed or a metaproperty operation is performed. Setting the SQL Server

ODBC driver-specific statement attribute SQL_SOPT_SS_DEFER_PREPARE to SQL_DP_OFF can turn off this default behavior.

In case of deferred prepare, calling either **SQLDescribeCol** or **SQLDescribeParam** before calling **SQLExecute** generates an extra roundtrip to the server. On **SQLDescribeCol**, the driver removes the WHERE clause from the query and sends it to the server with SET FMTONLY ON to get the description of the columns in the first result set returned by the query. On **SQLDescribeParam**, the driver calls the server to get a description of the expressions or columns referenced by any parameter markers in the query. This method also has some restrictions, such as not being able to resolve parameters in subqueries.

Excess use of **SQLPrepare** with the SQL Server ODBC driver degrades performance, especially when connected to earlier versions of SQL Server. Prepared execution should not be used for statements executed a single time. Prepared execution is slower than direct execution for a single execution of a statement because it requires an extra network roundtrip from the client to the server. On earlier versions of SQL Server it also generates a temporary stored procedure.

Prepared statements cannot be used to create temporary objects on SQL Server 2000, or on earlier versions of SQL Server if the option to generate stored procedures is active. With this option turned on, the prepared statement is built into a temporary stored procedure that is executed when **SQLExecute** is called. Any temporary object created during the execution of a stored procedure is automatically dropped when the procedure finishes. Either of the following examples results in the temporary table **#sometable** not being created if the option to generate stored procedures for prepare is active:

```
SQLPrepare(hstmt,
```

```
"CREATE TABLE #sometable(cola int, colb char(8))",
SQL_NTS);SQLExecute(hstmt);
```

or

```
SQLPrepare(hstmt,
"SELECT * FROM authors INTO #sometable",
```

SQL_NTS); SQLExecute(hstmt);

Some early ODBC applications used **SQLPrepare** anytime **SQLBindParameter** was used. **SQLBindParameter** does not require the use of **SQLPrepare**, it can be used with **SQLExecDirect**. For example, use **SQLExecDirect** with **SQLBindParameter** to retrieve the return code or output parameters from a stored procedure that is only executed one time. Do not use **SQLPrepare** with **SQLBindParameter** unless the same statement will be executed multiple times.

SQLPrepare on SQL Server version 6.5 or earlier

Earlier versions of SQL Server did not directly support prepared execution. To get the benefits of prepared execution on earlier versions of SQL Server, the SQL Server ODBC driver uses temporary stored procedures. On **SQLPrepare**, the SQL Server ODBC driver builds the SQL statement from the application into a CREATE PROCEDURE statement that it then sends to the server. This creates a temporary stored procedure and is essentially the same as having SQL Server parse the SQL statement and compile it into an execution plan. The names of the temporary stored procedures generated by the SQL Server ODBC driver start with #odbc#. On **SQLExecute**, the driver calls the stored procedure created on **SQLPrepare**. Administrators of SQL Server versions 6.0 or 6.5 must estimate the peak demand for **SQLPrepare** and make **tempdb** large enough to hold these temporary stored procedures.

SQL Server version 4.21a does not support temporary stored procedures. When connected to SQL Server 4.21a the SQL Server ODBC driver generates permanent stored procedures instead of temporary stored procedures. These permanent stored procedures are stored in the user databases, so administrators must ensure the user databases are large enough to hold the peak number of **SQLPrepare** functions. Also, the permanent stored procedures can be left in the database if the application terminates or loses its connection before the ODBC driver can drop the procedures. SQL Server 4.21a administrators may be required to periodically drop these stored procedures.

If an application will be run by many concurrent users and the users will all be using the same SQL statement, the best approach is to create the SQL statement as a permanent, parameterized stored procedure, and execute it with **SQLExecDirect**. Having many users issue concurrent **SQLPrepare** commands on earlier versions of SQL Server can create a concurrency problem on the system tables in **tempdb**. Even if each user is executing exactly the same statement, the SQL Server ODBC driver on each client is creating its own copy of a temporary stored procedure in **tempdb**. If the SQL statement is created as a parameterized stored procedure, however, the procedure is created only once. Each ODBC application does not have to create a new procedure for its exclusive use. It simply uses a copy of the execution plan of the permanent procedure from the procedure cache.

To avoid holding locks on **tempdb** system tables for the length of a user transaction, the SQL Server ODBC driver does not generate a stored procedure for **SQLPrepare** if it is called within a transaction. The exception to this is when the **SQLPrepare** is the first statement in the transaction. In this case, the driver generates a stored procedure but then immediately commits the CREATE PROCEDURE statement.

The driver does not generate a stored procedure for a **SQLPrepare** that uses the ODBC CALL escape clause to call a stored procedure. On **SQLExecute**, the driver executes the called stored procedure. (Creating a temporary stored procedure is not required.)

Whether the SQL Server ODBC driver generates temporary stored procedures when connected to earlier versions of SQL Server, and how long the procedures are retained, is controlled by data source parameters or connection attributes. The connection attributes are set by calling **SQLSetConnectAttr** with *fOption* set to SQL_COPT_SS_USE_PROC_FOR_PREPARE. The options are:

SQL_UP_OFF

Temporary stored procedures are not generated for **SQLPrepare**.

SQL_UP_ON

Temporary stored procedures are generated for **SQLPrepare** and are not dropped until the connection is closed. This is the default setting.

SQL_UP_ON_DROP

Temporary stored procedures are generated for **SQLPrepare**. The procedures are dropped the next time **SQLPrepare** is called on the statement

handle, when **SQLFreeHandle** is called to drop the statement handle, or when the connection is closed.

When SQL_UP_ON is set, most applications realize a performance boost because the SQL Server ODBC driver does not have to continually drop the temporary stored procedures. If an application reprepares an SQL statement when SQL_UP_ON_DROP is set, the driver can reuse the stored procedure created the first time the SQL statement was prepared. Applications that never disconnect (such as a 24x7 application) or that make heavy use of **SQLPrepare** can see a buildup of **#odbc#** procedures in **tempdb**. These applications should set SQL_UP_ON_DROP to alleviate the buildup.

Some APIs that map over ODBC (such as DAO) and the OLE DB Provider for ODBC do not expose the ability to set driver-specific connection attributes. Applications using these APIs cannot dynamically control the SQL_USE_PROC_FOR_PREPARE settings. If these applications use a SQL Server data source, these options can be set on the data source. This is done with the driver-specific **UseProcForPrepare** keyword on **SQLConfigDataSource**, or with the procedure options displayed in the SQL Server DSN Configuration Wizard.

To use a statement

Procedures

A stored procedure is a precompiled executable object that contains one or more Microsoft® SQL ServerTM statements. Stored procedures can have input and output parameters and can also put out an integer return code. An application can enumerate available stored procedures by using catalog functions.

ODBC applications the target SQL Server should only use direct execution to call a stored procedure. When connected to earlier versions of SQL Server, the SQL Server ODBC driver implements **SQLPrepare** by creating a temporary stored procedure, which is then called on **SQLExecute**. It adds overhead to have **SQLPrepare** create a temporary stored procedure that only calls the target stored procedure versus directly executing the target stored procedure. Even when connected to an instance of SQL Server, preparing a call requires an extra round trip across the network and the building of an execution plan that just calls the stored procedure execution plan.

ODBC applications should use the ODBC CALL syntax when executing a stored procedure. The driver is optimized to use a remote procedure call mechanism to call the procedure when the ODBC CALL syntax is used. This is more efficient than the mechanism used to send a Transact-SQL EXECUTE statement to the server.

For more information, see <u>Running Stored Procedures</u>.

To call remote procedures

Batches of Statements

A batch of Microsoft® SQL Server[™] statements contains two or more SQL statements, separated by a semicolon (;), built into a single string passed to **SQLExecDirect** or **SQLPrepare**. For example:

SQLExecDirect(hstmt, "SELECT * FROM authors; SELECT * FROM titles", SQL_NTS);

Batches can be more efficient than submitting statements separately because network traffic is often reduced. Use **SQLMoreResults** to get positioned on the next result set when finished with the current result set.

Batches can always be used when the ODBC cursor attributes are set to the defaults of a forward-only, read-only cursor with a rowset size of 1.

If a batch is executed when using server cursors against SQL Server, the server cursor is implicitly converted to a default result set. **SQLExecDirect** or **SQLExecute** return SQL_SUCCESS_WITH_INFO, and a call to **SQLGetDiagRec** returns:

```
szSqlState = "01S02", pfNativeError = 0
szErrorMsg = "[Microsoft][ODBC SQL Server Driver]Cursor type cha
```

Batches are not supported with server cursors against SQL Server version 6.5 or earlier. **SQLExecDirect** or **SQLExecute** return SQL_ERROR, and a call to **SQLGetDiagRec** returns one of three errors. For more information, see <u>Constructing SQL Statements for Cursors</u>.

See Also

SQLMoreResults
SQLPrepare

Effects of SQL-92 Options

The ODBC standard is closely matched to the SQL-92 standard, and ODBC applications expect standard behavior from an ODBC driver. To make its behavior conform more closely with that defined in the ODBC standard, the Microsoft® SQL Server[™] ODBC driver always uses any SQL-92 options available in the version of SQL Server with which it connects.

When the SQL Server ODBC driver connects to an instance of SQL Server, the server detects that the client is using the ODBC driver and sets several options on. The options set on by SQL Server 2000 are the same as those turned on by SET statements when the driver connects to an instance of SQL Server version 6.5, except that SQL Server 2000 also sets on the CONCAT_NULL_YIELDS_NULL option.

The options set by the driver when connecting to each prior version of SQL Server are:

• Connect to an instance of SQL Server 6.5:

SET QUOTED_IDENTIFIER ON SET TEXTSIZE 2147483647 SET ANSI_DEFAULTS ON SET CURSOR_CLOSE_ON_COMMIT OFF SET IMPLICIT_TRANSACTIONS OFF

• Connect to an instance of SQL Server 6.0:

SET ANSI_NULL_DFLT_ON ON SET TEXTSIZE 2147483647 SET QUOTED_IDENTIFIER ON SET ARITHABORT ON

• Connect to an instance of SQL Server 4.21a:

SET TEXTSIZE 2147483647 SET ARITHABORT ON

The driver issues these statements itself; the ODBC application does nothing to

request them. Setting these options allows ODBC applications using the driver to be more portable because the server behavior then matches the SQL-92 standard.

DB-Library-based applications generally do not turn these options on. Sites observing different behavior between ODBC or DB-Library clients when running the same SQL statement should not assume this points to a problem with the ODBC driver. They should first rerun the statement in the DB-Library environment with the same SET options as would be used by the SQL Server ODBC driver.

Because SET options can be turned on and off at any time by users and applications, developers of stored procedures and triggers should also take care to test their procedures and triggers with the SET options listed above turned both on and off. This ensures that the procedures and triggers work correctly regardless of which options a particular connection may have set on when they invoke the procedure or trigger. Triggers or stored procedures that require a particular setting for one of these options should issue a SET statement at the start of the trigger or stored procedure. This SET statement remains in effect only for the execution of the trigger or stored procedure; when the procedure or trigger ends, the original setting is restored.

The SET options used when connected to SQL Server 7.0 or SQL Server 6.5 have the net effect of setting on three more SQL-92 options than those set in the 6.0 environment: ANSI_NULLS, ANSI_PADDING, and ANSI_WARNINGS. When connected to an instance of SQL Server 2000, a fourth option, CONCAT_NULL_YIELDS_NULL, is also set on. These options can cause problems in existing stored procedures and triggers migrated from SQL Server 6.0 to either SQL Server 6.5 or 7.0. The SQL Server ODBC driver does not set these options on if AnsiNPW=NO is specified in the data source or on either **SQLDriverConnect** or **SQLBrowseConnect**.

The SQL Server ODBC driver also sets on the QUOTED_IDENTIFIER option when connected to SQL Server 6.0 or later. With this option set on, SQL statements should comply with the SQL-92 rule that character data strings be enclosed in single quotes and that only identifiers, such as table or column names, be enclosed in double quotation marks:

SELECT "au_fname" FROM "authors"

WHERE "au_lname" = 'O"Brien'

Like the SQL-92 options noted earlier, the SQL Server ODBC driver does not turn the QUOTED_IDENTIFIER option on if QuotedID=NO is specified in the data source or on either **SQLDriverConnect** or **SQLBrowseConnect**.

To allow the driver to know the current state of SET options, ODBC applications should not use the Transact-SQL SET statement to set these options. They should only set these options using either the data source or the connection options. If the application issues SET statements, the driver can generate incorrect SQL statements.

See Also

<u>SQLBrowseConnect</u>

SQLDriverConnect

Freeing a Statement Handle

As mentioned earlier, it is more efficient to reuse statement handles than drop them and allocate new ones. Before executing a new Microsoft® SQL Server[™] statement on a statement handle, applications should check that the current statement settings are appropriate. These include statement attributes, parameter bindings, and result set bindings. Generally, parameters and result sets for the old SQL statement must be unbound (by calling **SQLFreeStmt** with the SQL_RESET_PARAMS and SQL_UNBIND options) and rebound for the new SQL statement.

When the application has finished using the statement, it calls **SQLFreeHandle** to free the statement. Note that **SQLDisconnect** automatically frees all statements on a connection.

To use a statement

Processing Results

After an application submits an SQL statement, Microsoft® SQL ServerTM returns any resulting data as one or more result sets. A result set is a set of rows and columns that match the criteria of the query. SELECT statements, catalog functions, and some procedures produce a result set made available to an application in tabular form. If the executed SQL statement is a stored procedure, a batch containing multiple commands, or a SELECT statement containing keywords, such as COMPUTE or COMPUTE BY, there will be multiple result sets to process.

ODBC catalog functions also can retrieve data. For example, **SQLColumns** retrieves data about columns in the data source. These result sets can contain zero or more rows.

Note that other SQL statements, such as GRANT or REVOKE, do not return result sets. For these statements, the return code from **SQLExecute** or **SQLExecDirect** is usually the only indication the statement was successful.

Each INSERT, UPDATE, and DELETE statement returns a result set containing only the number of rows affected by the modification. This count is made available when application calls **SQLRowCount**. ODBC 3.*x* applications must either call **SQLRowCount** to retrieve the result set or **SQLMoreResults** to cancel it. When an application executes a batch or stored procedure containing multiple INSERT, UPDATE, or DELETE statements, the result set from each modification statement must be processed using **SQLRowCount** or cancelled using **SQLMoreResults**. These counts can be cancelled by including a SET NOCOUNT ON statement in the batch or stored procedure.

Transact-SQL includes the SET NOCOUNT statement. When the NOCOUNT option is set on, SQL Server does not return the counts of the rows affected by a statement and **SQLRowCount** returns 0. The SQL Server ODBC driver version 3.7 introduces a driver-specific **SQLGetStmtAttr** option,

SQL_SOPT_SS_NOCOUNT_STATUS, to report on whether the NOCOUNT option is on or off. Anytime **SQLRowCount** returns 0, the application should test SQL_SOPT_SS_NOCOUNT_STATUS. If SQL_NC_ON is returned, the value of 0 from **SQLRowCount** only indicates that SQL Server has not returned a row count. If SQL_NC_OFF is returned, it means that NOCOUNT is off and

the value of 0 from **SQLRowCount** indicates that the statement did not affect any rows. Applications should not display the value of **SQLRowCount** when SQL_SOPT_SS_NOCOUNT_STATUS is SQL_NC_OFF. Large batches or stored procedures may contain multiple SET NOCOUNT statements so programmers cannot assume SQL_SOPT_SS_NOCOUNT_STATUS remains constant. The option should be tested each time **SQLRowCount** returns 0.

Several other Transact-SQL statements return their data in messages rather than result sets. When the SQL Server ODBC driver receives these messages, it returns SQL_SUCCESS_WITH_INFO to let the application know that informational messages are available. The application can then call **SQLGetDiagRec** to retrieve these messages. The Transact-SQL statements that work this way are:

- DBCC
- SET SHOWPLAN (available with earlier versions of SQL Server)
- SET STATISTICS
- PRINT
- RAISERROR

The SQL Server ODBC driver returns SQL_ERROR on a RAISERROR with a severity of 11 or higher. If the severity of the RAISERROR is 19 or higher, the connection is also dropped.

To process the result sets from an SQL statement, the application:

- Determines the characteristics of the result set.
- Binds the columns to program variables.
- Retrieves a single value, an entire row of values, or multiple rows of values.

• Tests to see if there are more result sets, and if so, loops back to determining the characteristics of the new result set.

The process of retrieving rows from the data source and returning them to the application is called fetching.

Retrieving COMPUTE and COMPUTE BY result sets

The COMPUTE BY clause generates subtotals within a result set; the COMPUTE clause generates a total at the end of the result set. The SQL Server ODBC driver presents these totals and subtotals to the calling application by generating multiple result sets for each SELECT statement.

The following example uses COMPUTE BY to generate subtotals and COMPUTE to generate a total:

SELECT title = CONVERT(char(20), title), type, price, advance FROM titles WHERE ytd_sales IS NOT NULL AND type LIKE '%cook%' ORDER BY type DESC COMPUTE AVG(price), SUM(advance) BY type COMPUTE SUM(price), SUM(advance)

These statements cause a subtotal calculation for the average price and sum of advances for each book type and then cause a final total sum of both the price and advance data. The driver presents the first result set for the rows from books having the first book type. It then produces a second result set with the two COMPUTE BY columns for the AVG(price) and SUM(advance) for this first set of books. Then it produces a third result set for the next group of books, and a fourth result set with the COMPUTE BY subtotals for that group. The driver interleaves these result sets until it produces the final result set with the total for the COMPUTE SUM(price), SUM(advance) clause.

See Also

SQLColumns
SQLRowCount

Determining the Characteristics of a Result Set

Meta data is data that describes other data. For example, result set meta data describes the characteristics of a result set, such as the number of columns in the result set, the data types of those columns, their names, precision, nullability, and so on.

ODBC supplies meta data to applications through its catalog API functions. The Microsoft® SQL ServerTM ODBC driver implements many of the ODBC API catalog functions as calls to a corresponding SQL Server catalog procedure.

Applications require meta data for most result set operations. For example, the application uses the data type of a column to determine what kind of variable to bind to that column. It uses the byte length of a character column to determine how much space it needs to display data from that column. How an application determines the meta data for a column depends on the type of the application.

Vertical applications typically work with predefined tables and perform predefined operations on those tables. Because the result set meta data for such applications is defined before the application is even written and is controlled by the application developer, it can be hard-coded into the application. For example, if an order ID column is defined as a 4-byte integer in the data source, the application can always bind a 4-byte integer to that column. When meta data is hard-coded in the application, a change to the tables used by the application generally implies a change to the application code.

Generic applications, especially applications that support ad hoc queries, almost never know the meta data of the result sets they create. Therefore, they must discover the meta data at run time.

To determine the characteristics of a result set, an application can call:

- **SQLNumResultCols** to determine how many columns a request returned.
- **SQLColAttribute** or **SQLDescribeCol** to describe a column in the result set.

A well-designed application is written with the assumption that the result set is unknown and uses the information returned by these functions to bind the columns in the result set. An application can call these functions at any time after a statement is prepared or executed. However, for optimal performance, an application should call **SQLColAttribute**, **SQLDescribeCol**, and **SQLNumResultCols** after a statement is executed.

You can have multiple concurrent calls for meta data. The system catalog procedures underlying the ODBC catalog API implementations can be called by the ODBC driver while it is using static server cursors. This allows applications to concurrently process multiple calls to ODBC catalog functions.

If an application uses a particular set of meta data more than once, it will probably benefit by caching the information in private variables when it is first obtained. This eliminates the overhead of later calls to the ODBC catalog functions for the same information (which forces the driver to make roundtrips to the server).

To retrieve result set information

Assigning Storage (Binding)

An application can assign storage for results before or after it executes an SQL statement. If an application prepares or executes the SQL statement first, it can inquire about the result set before it assigns storage for results. For example, if the result set is unknown, the application must retrieve the number of columns before it can assign storage for them.

To associate storage for a column of data, an application calls **SQLBindCol** and passes it:

- The data type to which the data is to be converted.
- The address of an output buffer for the data.

The application must allocate this buffer, and it must be large enough to hold the data in the form to which it is converted.

• The length of the output buffer.

This value is ignored if the returned data has a fixed width in C, such as an integer, real number, or date structure.

• The address of a storage buffer in which to return the number of bytes of available data.

An application can also bind result set columns to arrays of program variables to support fetching result set rows in blocks. There are two different types of array binding:

• Column-wise binding is done when each individual column is bound to its own array of variables.

Column-wise binding is specified by calling **SQLSetStmtAttr** with *Attribute* set to SQL_ATTR_ROW_BIND_TYPE and *ValuePtr* set to SQL_BIND_BY_COLUMN. All of the arrays must have the same number of elements.

• Row-wise binding is done when all of the parameters in the SQL

statement are bound as a unit to an array of structures that contain the individual variables for the parameters.

Row-wise binding is specified by calling **SQLSetStmtAttr** with *Attribute* set to SQL_ATTR_ROW_BIND_TYPE and *ValuePtr* set to the size of the structure holding the variables that will receive the result set columns.

The application also sets SQL_ATTR_ROW_ARRAY_SIZE to the number of elements in the column or row arrays, and sets SQL_ATTR_ROW_STATUS_PTR and SQL_ATTR_ROWS_FETCHED_PTR.

To process results

Fetching Result Data

An ODBC application has three options for fetching result data.

The first option is based on **SQLBindCol**. Before fetching the result set, the application uses **SQLBindCol** to bind each column in the result set to a program variable. After the columns have been bound, the driver transfers the data of the current row into the variables bound to the result set columns each time the application calls **SQLFetch** or **SQLFetchScroll**. The driver handles data conversions if the result set column and program variable have different data types. If the application has SQL_ATTR_ROW_ARRAY_SIZE set greater than 1, it can bind result columns to arrays of variables, which will all be filled on each call to **SQLFetchScroll**.

The second option is based on **SQLGetData**. The application does not use **SQLBindCol** to bind result set columns to program variables. After each call to **SQLFetch**, the application calls **SQLGetData** once for each column in the result set. **SQLGetData** instructs the driver to transfer data from a specific result set column to a specific program variable and specifies the data types of the column and variable. This allows the driver to convert data if the result column and program variable have different data types. Text, ntext, and image columns are typically too large to fit into a program variable but can still be retrieved using **SQLGetData**. If the **text**, **ntext**, or **image** data in the result column is larger than the program variable, **SQLGetData** returns SQL SUCCESS WITH INFO and SQLSTATE 01004 (string data, right truncated). Successive calls to SQLGetData return successive chunks of the text or image data. When the end of the data is reached, SQLGetData returns SQL SUCCESS. Each fetch returns a set of rows, or rowset, if SQL_ATTR_ROW_ARRAY_SIZE is greater than 1. Before using **SQLGetData**, you must first use **SQLSetPos** to specify a specific row within the rowset as the current row.

The third option is to use a mix of **SQLBindCol** and **SQLGetData**. An application could, for example, bind the first ten columns of a result set and then, on each fetch, call **SQLGetData** three times to retrieve the data from three unbound columns. This would typically be used when a result set contains one or more **text** or **image** columns.

Depending on the cursor options set for the result set, an application can also use the scrolling options of **SQLFetchScroll** to scroll around the result set.

Excess use of **SQLBindCol** to bind a result set column to a program variable is expensive because **SQLBindCol** causes an ODBC driver to allocate memory. When you bind a result column to a variable, that binding remains in effect until you either call **SQLFreeHandle** to free the statement handle or call **SQLFreeStmt** with *fOption* set to SQL_UNBIND. The bindings are not automatically undone when the statement completes.

This logic allows you to effectively deal with executing the same SELECT statement several times with different parameters. Because the result set keeps the same structure, you can bind the result set once, process all the SELECT statements, then call **SQLFreeStmt** with *fOption* set to SQL_UNBIND after the last execution. You should not call **SQLBindCol** to bind the columns in a result set without first calling **SQLFreeStmt** with *fOption* set to SQL_UNBIND to free any previous bindings.

When using **SQLBindCol**, you can either do row-wise or column-wise binding. Row-wise binding is somewhat faster than column-wise binding.

You can use **SQLGetData** to retrieve data on a column-by-column basis instead of binding result set columns using **SQLBindCol**. If a result set contains only a few rows, using **SQLGetData** instead of **SQLBindCol** is faster; otherwise, **SQLBindCol** gives the best performance. If you do not always put the data in the same set of variables, you should use **SQLGetData** instead of constantly rebinding. You can only use **SQLGetData** on columns that are in the select list after all columns are bound with **SQLBindCol**. The column must also appear after any columns on which you have already used **SQLGetData**.

The ODBC functions that deal with moving data into or out of program variables, such as **SQLGetData**, **SQLBindCol**, and **SQLBindParameter**, support implicit data type conversion. For example, if an application binds an integer column to a character string program variable, the driver automatically converts the data from integer to character before placing it into the program variable.

Data conversion in applications should be minimized. Unless data conversion is required for the processing done by the application, applications should bind columns and parameters to program variables of the same data type. If the data must be converted from one type to another, however, it is more efficient to have the driver do the conversion than doing it in the application. The Microsoft® SQL Server[™] ODBC driver normally just transfers data directly from the network buffers to the variables of the application. Requesting the driver to do data conversion forces the driver to buffer the data and use CPU cycles to convert the data.

Program variables should be large enough to hold data transferred in from a column, except for **text**, **ntext**, and **image** data. If an application attempts to retrieve result set data and place it into a variable that is too small to hold it, the driver generates a warning. This forces the driver to allocate memory for the message, and the driver and application both have to spend CPU cycles processing the message and doing error handling. The application should either allocate a variable large enough to hold the data being retrieved or use the SUBSTRING function in the select list to reduce the size of the column in the result set.

Care must be taken when using SQL_C_DEFAULT to specify the type of the C variable. SQL_C_DEFAULT specifies that the type of the C variable matches the SQL data type of the column or parameter. If SQL_C_DEFAULT is specified for an **ntext**, **nchar**, or **nvarchar** column, Unicode data is returned to the application. This can cause various problems if the application has not been coded to handle Unicode data. The same types of problems can occur with the **uniqueidentifier** (SQL_GUID) data type. In these cases, use the **odbccmpt** utility to set the 6.5 ODBC compatibility option until the application can be changed. With the 6.5 ODBC compatibility option, Unicode data is converted to character and **uniqueidentifier** is converted to **varbinary**.

text, **ntext**, and **image** data is typically too large to fit into a single program variable, and is usually processed with **SQLGetData** instead of **SQLBindCol**. When using server cursors, the SQL Server ODBC driver is optimized to not transmit the data for unbound **text**, **ntext**, or **image** columns at the time the row is fetched. The **text**, **ntext**, or **image** data is not actually retrieved from the server until the application issues **SQLGetData** for the column.

This optimization can be applied to applications so that no **text**, **ntext**, or **image** data is displayed while a user is scrolling up and down a cursor. After the user selects a row, the application can call **SQLGetData** to retrieve the **text**, **ntext**, or **image** data. This saves transmitting the **text**, **ntext**, or **image** data for any of the

rows the user does not select and can save the transmission of very large amounts of data.

To process results

Mapping Data Types

The Microsoft® SQL Server[™] ODBC driver maps SQL Server SQL data types to ODBC SQL data types. The illustration below shows SQL Server SQL data types and the ODBC SQL data types to which they map. It also shows ODBC SQL data types and their corresponding ODBC C data types, and the supported and default conversions.

							1	C d	ata	typ	e				
SQL Server data type	Default ODBC data type	SQL_C_CHAR	SQL_C_WCHAR	SQL_C_BIT	SQL_C_NUMERIC	SQL_C_TINYINT	SQL_C_SHORT	SQL_C_LONG	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_BINARY	SQL_C_TYPE_TIMESTAMP	SQL_C_TYPE_DATE	SQL_C_TYPE_TIME	sql_c_guid
binary	SQL_BINARY	00	00								•				
varbinary	SQL_VARBINARY SQL_CHAR	R	0	0	0	0	0	0		0		~	~	~	
char nchar	SQL_CHAR SQL WCHAR	X	K	×	0000	0000	0	0000	0000	0000	0000	0000	0000	000	00
nchar varchar	SQL_WCHAN	0.0.0	-	000	S	R	×	×	×	×	R	×	×	R	ŏ
sysname	SQL_VARCHAR	H	00	X	×	×	00	X	×	×	X	×	×	ŏ	\sim
nvarchar	SQL WVARCHAR		ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	ŏ	0
datetime	SQL_TYPE_TIMESTAMP	ŏ	ō	Ť	Ť	-	-	-	Ť	\sim	ŏ	ě	ŏ	ŏ	\ge
smalldatetime	SQL_TYPE_TIMESTAMP	ŏ	ŏ				-				ŏ	ŏ	ŏ	ŏ	
decimal	SQL DECIMAL	ŏ	ŏ	0	•	0	0	0	0	0	ŏ		Ť	-	
numeric	SQL NUMERIC	ō	Ō	Ō	Ō	00	00	00	000	Õ	ŏ				
float	SQL_FLOAT	Ō	0	0	0	0	0	0	Ō	•	ŏ				
real	SQL_REAL	0	0	0	000	0	0	0	٠	0	Ō				
int	SQL_INTEGER	0	0	0	0	0	0	٠	00	0	0				
smallint	SQL_SMALLINT	\circ	0	\circ	0	0	٠	\circ	0	0	0				
tinyint	SQL_TINYINT	\circ	0	\circ	0	٠	0	0	0	0	0				
money	SQL_DECIMAL	0	0	\circ	٠	0	0	0	\circ	0	0				
smallmoney	SQL_DECIMAL		0	0	٠	0	0	0	0	0	0				
bit	SQL_BIT	0	0		0	0	0	0	0	0	0				
timestamp	SQL_BINARY	0	0								٠				
uniqueidentifier	SQL_GUID	0									0				•
image	SQL_LONGVARBINARY	0	0	0	0	0	0	0	0	0	•	0	0	0	
ntext	SQL_WLONGVARCHAR	0		0	0	0	0	0	0	0	0	0	0	0	
text	SQL_LONGVARCHAR	•	0	0	0	0	0	0	0	0	0	0	0	0	

Note The SQL Server **timestamp** data type maps to the SQL_BINARY or SQL_VARBINARY ODBC data type because the values in **timestamp** columns are not **datetime** values, but **binary(8)** or **varbinary(8)** values that indicate the sequence of SQL Server activity on the row. If the SQL Server ODBC driver encounters a SQL_C_WCHAR (Unicode) value that is an odd number of bytes,

the trailing odd byte is truncated.

Dealing with sql_variant data type in ODBC

The **sql_variant** data type column can contain any of the data types in SQL Server except large objects (LOBs), such as text, ntext, image. For example, the column could contain smallint values for some rows, float values for other rows, and char/nchar values in the remainder.

The **sql_variant** data type is similar to the variant data type in Microsoft Visual Basic[®].

Retrieving Data from the Server

ODBC does not have a notion of variant types. This limits the use of the **sql_variant** data type with an ODBC driver in SQL Server 2000. In SQL Server 2000, if binding is specified, the **sql_variant** data type must be bound to one of the documented ODBC data types. **SQL_CA_SS_VARIANT_TYPE**, a new attribute specific to the SQL Server ODBC driver, returns the data type of an instance in the **sql_variant** column to the user.

If no binding is specified, the **SQLGetData** function can be used to determine the data type of an instance in the **sql_variant** column.

To retrieve **sql_variant** data follow these steps.

- 1. Call **SQLFetch** to position to the row retrieved.
- Call SQLGetData, specifying SQL_C_BINARY for the type and 0 for the data length. This forces the driver to read the sql_variant header. The header provides the data type of that instance in the sql_variant column. SQLGetData returns the size (in bytes) of the value.
- 3. Call **SQLColAttribute** by specifying **SQL_CA_SS_VARIANT_TYPE** as its attribute value. This function will return the C data type of the instance in the **sql_variant** column to the client.

Here is a code segment showing the preceding steps.

```
while ((retcode = SQLFetch (hstmt))==SQL_SUCCESS)
ł
 if (retcode != SQL_SUCCESS && retcode != SQL_SUCCESS_WI7
 {
   SQLError (NULL, NULL, hstmt, NULL,
        &lNativeError,szError,MAX DATA,&sReturned);
   printf ("%s\n",szError);
   goto Exit;
 }
 retcode = SQLGetData (hstmt, 1, SQL_C_BINARY,
             pBuff,0,&Indicator);//Figure out the length
 if (retcode != SQL SUCCESS WITH INFO && retcode != SQL S
 {
   SQLError (NULL, NULL, hstmt, NULL, &lNativeError,
        szError,MAX DATA,&sReturned);
   printf ("%s\n",szError);
   goto Exit;
 }
 printf ("Byte length : %d ",Indicator); //Print out the byte length
 int iValue = 0;
 retcode = SQLColAttribute (hstmt, 1, SQL_CA_SS_VARIANT_TY)
                 NULL,NULL,&iValue); //Figure out the type
 printf ("Sub type = %d ",iValue);//Print the type, the return is C_type
//Set up a new binding or do the SQLGetData on that column with
//the appropriate type
```

}

If the user creates the binding using **SQLBindCol**, the driver reads the meta data and the data. The driver then converts the data to the appropriate ODBC type specified in the binding.

Sending Data to the Server

SQL_SS_VARIANT, a new data type specific to the SQL Server ODBC driver, is used for data sent to an **sql_variant** column. When sending data to the server using parameters (for example, INSERT INTO TableName VALUES (?,?)), **SQLBindParameter** is used to specify the parameter information including the C type and the corresponding SQL Server type. The SQL Server ODBC driver will convert the C data type to one of the appropriate **sql_variant** subtypes.

Data Type Usage

The Microsoft® SQL Server[™] ODBC driver and SQL Server impose the following use of data types.

Data type	Limitation
Date literals	Date literals, when stored in a SQL_TYPE_TIMESTAMP column (SQL Server data types of datetime or smalldatetime), have a time value of 12:00:00.000 A.M.
money and smallmoney	Only the integer parts of the money and smallmoney data types are significant. If the decimal part of SQL money data is truncated during data type conversion, the SQL Server ODBC driver returns a warning, not an error.
SQL_BINARY (nullable)	When connected to an instance of SQL Server version 6.0 and earlier, if a SQL_BINARY column is nullable, the data that is stored in the data source is not padded with zeroes. When data from such a column is retrieved, the SQL Server ODBC driver pads it with zeroes on the right. However, data that is created in operations performed by SQL Server, such as concatenation, does not have such padding.
	Also, when data is placed in such a column in an instance of SQL Server 6.0 or earlier, SQL Server truncates the data on the right if it is too long to fit into the column.
SQL_CHAR (truncation)	When connected to an instance of SQL Server 6.0 and earlier, and data is placed into a SQL_CHAR column, SQL Server truncates it on the right without warning if the data is too long to fit into the column.

	string_exp parameters to the string functions must be of data type SQL_CHAR or SQL_VARCHAR. SQL_LONG_VARCHAR data types are not supported in the string functions. The <i>count</i> parameter must be less than or equal to 8,000 because the SQL_CHAR and SQL_VARCHAR data types are limited to a maximum length of 8,000 characters. When connected to an instance of SQL Server 6.5 or earlier, the limit is 255 instead of 8000. Time literals, when stored in a SQL_TIMESTAMP column (SQL Server data types of datetime or smalldatetime), have a
	SQL_LONGVARBINARY, SQL_LONGVARCHAR, or SQL_WLONGVARCHAR data types (using a WHERE clause) that affect multiple rows are fully supported when connected to an instance of SQL Server 6. <i>x</i> and later. When connected to an instance of SQL Server 4.2 <i>x</i> , an S1000 error "Partial insert/update. The insert/update of a text or image column(s) did not succeed" is returned if the update affects more than one
SQL_CHAR (nullable)	When connected to an instance of SQL Server 6.0 and earlier, if a SQL_CHAR column is nullable, the data that is stored in the data source is not padded with blanks. When data from such a column is retrieved, the SQL Server ODBC driver pads it with blanks on the right. However, data that is created in operations performed by SQL Server, such as concatenation, does not have such padding.

tinyint	 into a timestamp column. However, because timestamp columns are automatically updated by SQL Server, a NULL value is overwritten. The SQL Server tinyint data type is unsigned. A tinyint column is bound to a variable of data type SQL_C_UTINYINT by default.
User-defined data types	 When connected to an instance of SQL Server 4.2<i>x</i>, the SQL Server ODBC driver adds NULL to a column definition that does not explicitly declare a column's nullability. Therefore, the nullability that is stored in the definition of a user-defined data type is ignored.
	 When connected to an instance of SQL Server 4.2<i>x</i>, columns with a user-defined data type that has a base data type of char or binary and for which no nullability is declared are created as data type varchar or varbinary. SQLColAttribute, SQLColumns, and SQLDescribeCol return SQL_VARCHAR or SQL_VARBINARY as the data type for these columns. Data that is retrieved from these columns is not padded.
LONG data types	<i>data-at-execution</i> parameters are restricted for both the SQL_LONGVARBINARY and the SQL_LONGVARCHAR data types.

Autotranslation of Character Data

Character data, such as ANSI character variables declared with SQL_C_CHAR or data stored in Microsoft® SQL Server[™] using the **char**, **varchar**, or **text** data types, can represent only a limited number of characters. Character data stored using one byte per character can only represent 256 characters. The values stored in SQL_C_CHAR variables are interpreted using the ANSI code page (ACP) of the client computer. The values stored using **char**, **varchar**, or **text** data types on the server are evaluated using the ACP of the server.

If both the server and the client have the same ACP, then they have no problems in interpreting the values stored in SQL_C_CHAR, **char**, **varchar**, or **text** objects. If the server and client have different ACPs, then SQL_C_CHAR data from the client may be interpreted as a different character on the server if it is used in **char**, **varchar**, or **text** columns, variables, or parameters. For example, a character byte containing the value 0xA5 is interpreted as the character \tilde{N} on a computer using code page 437 and is interpreted as the yen sign (¥) on a computer running code page 1252.

Unicode data is stored using two bytes per character. All extended characters are covered by the Unicode specification, so all Unicode characters are interpreted the same by all computers.

The AutoTranslate feature of the SQL Server ODBC driver attempts to minimize the problems in moving character data between a client and a server that have different code pages. AutoTranslate can be set in the connect string of **SQLDriverConnect**, in the configuration string of **SQLConfigDataSource**, or when configuring data sources for the SQL Server ODBC driver using ODBC Administrator.

When AutoTranslate is no, no conversions are done on data moved between SQL_C_CHAR variables on the client and **char**, **varchar**, or **text** columns, variables, or parameters in a SQL Server database. The bit patterns may be interpreted differently on the client and server computers if the data contains extended characters and the two computers have different code pages. The data will be interpreted the same if both computers have the same code page.

When AutoTranslate is yes, the ODBC driver uses Unicode to convert data

moved between SQL_C_CHAR variables on the client and **char**, **varchar**, or **text** columns, variables, or parameters in a SQL Server database:

- When data is sent from an SQL_C_CHAR variable on the client to a **char**, **varchar**, or **text** column, variable, or parameter in an SQL Server database, the ODBC driver first converts from SQL_C_CHAR to Unicode using the ACP of the client, then from Unicode back to character using the ACP of the server.
- When data is sent from a **char**, **varchar**, or **text** column, variable, or parameter in a SQL Server database to a SQL_C_CHAR variable on the client, the ODBC driver first converts from character to Unicode using the ACP of the server, then from Unicode back to SQL_C_CHAR using the ACP of the client.

Because all of these conversions are done by the SQL Server ODBC driver executing on the client, the server ACP must be one of the code pages installed on the client computer.

Making the character conversions through Unicode ensures the proper conversion of all characters that exist in both code pages. If a character exists in one code page but not another, however, then the character cannot be represented in the target code page. For example, code page 1252 has the registered trademark symbol (®), while code page 437 does not.

The AutoTranslate setting has no effect on these conversions:

- Moving data between character SQL_C_CHAR client variables and Unicode **nchar**, **nvarchar**, or **ntext** columns, variables, or parameters in SQL Server databases.
- Moving data between Unicode SQL_C_WCHAR client variables and character **char**, **varchar**, or **text** columns, variables, or parameters in SQL Server databases.

Data always must be converted when moved from character to Unicode.

See Also

Collations

<u>SQLConfigDataSource</u>

<u>SQLDriverConnect</u>

Using Cursors

ODBC supports a cursor model that allows:

- Several types of cursors.
- Scrolling and positioning within a cursor.
- Several concurrency options.
- Positioned updates.

ODBC applications rarely declare and open cursors or use any cursor-related Transact-SQL statements. ODBC automatically opens a cursor for every result set returned from an SQL statement. The characteristics of the cursors are controlled by statement attributes set with **SQLSetStmtAttr** before the SQL statement is executed. The ODBC API functions for processing result sets support the full range of cursor functionality, including fetching, scrolling, and positioned updates.

This is a comparison of how Transact-SQL scripts and ODBC applications work with cursors.

Action	Transact-SQL	ODBC
Define cursor behavior	Specify through	Set cursor attributes by
	DECLARE CURSOR	using SQLSetStmtAttr
	parameters	
Open a cursor	DECLARE CURSOR	SQLExecDirect or
	OPEN cursor_name	SQLExecute
Fetch rows	FETCH	SQLFetch or
		SQLFetchScroll
Positioned update	WHERE CURRENT OF	SQLSetPos
	clause on UPDATE or	
	DELETE	

Close a cursor	CLOSE cursor_name	SQLCloseCursor
	DEALLOCATE	

The server cursors implemented in Microsoft® SQL Server[™] support the functionality of the ODBC cursor model. The SQL Server ODBC driver uses server cursors to support the cursor functionality of the ODBC API.

See Also

<u>CLOSE</u> <u>Cursors</u>

DEALLOCATE

DECLARE CURSOR

<u>FETCH</u>

<u>OPEN</u>

SQLCloseCursor

SQLFetchScroll

<u>SQLSetStmtAttr</u>

How Cursors Are Implemented

ODBC applications control the behavior of a cursor by setting one or more statement attributes before executing an SQL statement. ODBC has two different ways to specify the characteristics of a cursor:

• Cursor type

Cursor types are set using the SQL_ATTR_CURSOR_TYPE attribute of **SQLSetStmtAttr**. The ODBC cursor types are forward-only, static, keyset-driven, mixed, and dynamic. Setting the cursor type was the original method of specifying cursors in ODBC.

• Cursor behavior

Cursor behavior is set using the SQL_ATTR_CURSOR_SCROLLABLE and SQL_ATTR_CURSOR_SENSITIVITY attributes of **SQLSetStmtAttr**. These attributes are modeled on the SCROLL and SENSITIVE keywords defined for the DECLARE CURSOR statement in SQL-92 and ISO SQL standards. These two SQL-92/ISO options were introduced in ODBC version 3.0.

The characteristics of an ODBC cursor should be specified using either one or the other of these two methods, with the preference being to use the ODBC cursor types. While you can set all three options for the same cursor, the *Microsoft ODBC 3.0 Programmer's Reference* warns that this can prevent an ODBC driver from being able to efficiently implement the cursor.

In addition to setting the type of a cursor, ODBC applications also set other options, such as the number of rows returned on each fetch, concurrency options, and transaction isolation levels. These options can be set for either ODBC-style cursors (forward-only, static, keyset-driven, mixed, and dynamic) or SQL-92/ISO style cursors (scrollability and sensitivity).

The Microsoft® SQL Server[™] ODBC driver supports several ways to physically implement the various types of cursors. The driver implements some types of cursors using a SQL Server default result set; it implements others as server cursors or by using the ODBC Cursor Library.

To use cursors

Using Default Result Sets

The default ODBC cursor attributes are:

```
SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSO
SQLSetStmtAttr(hstmt, SQL_ATTR_CONCURRENCY, SQL_CONC
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, 1);
```

Whenever these attributes are set to their defaults, the Microsoft® SQL ServerTM ODBC driver uses a SQL Server default result set. Default result sets can be used for any SQL statement supported by SQL Server, and are the most efficient method of transferring an entire result set to the client.

Default result sets do not support multiple active statements on the same connection. After an SQL statement is executed on a connection, the server does not accept commands (except a request to cancel the rest of the result set) from the client on that connection until all the rows in the result set have been processed. To cancel the remainder of a partially processed result set, call **SQLCloseCursor** or **SQLFreeStmt** with the *fOption* parameter set to SQL_CLOSE. To finish a partially processed result set and test for the presence of another result set, call **SQLMoreResults**. If an ODBC application attempts a command on a connection handle before a default result set has been completely processed, the call generates SQL_ERROR and a call to **SQLGetDiagRec** returns:

szSqlState: "HY000", pfNativeError: 0 szErrorMsg: "[Microsoft][SQL Server ODBC Driver] Connection is busy with results for another hstmt."

To use cursors

Using Server Cursors

If an ODBC application sets any of the ODBC cursor attributes to anything other than the defaults, then the Microsoft® SQL Server[™] ODBC driver requests the server to implement an API server cursor of the same type. The use of API server cursors frees memory on the client and can significantly reduce network traffic between the client and server.

A potential drawback of API server cursors is that they currently do not support all SQL statements. API server cursors cannot be used to execute:

- Batches or stored procedures that return multiple result sets.
- SELECT statements that contain COMPUTE, COMPUTE BY, FOR BROWSE, or INTO clauses.
- An EXECUTE statement referencing a remote stored procedure.

When connected to an instance of SQL Server 2000, attempting to execute a statement with these characteristics using a server cursor results in the cursor being converted to a default result set. When connected to earlier versions of SQL Server, the attempt results in an error.

See Also

Cursor Implementations

ODBC Cursor Library

Some ODBC drivers only support the default cursor settings; these drivers also do not support positioned cursor operations, such as **SQLSetPos**. The ODBC cursor library is a component of the ODBC SDK used to implement block or static cursors on a driver that normally does not support them. The cursor library also implements positioned UPDATE and DELETE statements and **SQLSetPos** for the cursors it creates.

The ODBC cursor library is implemented as a layer between the ODBC Driver Manager and an ODBC driver. If the ODBC cursor library is loaded, the ODBC Driver Manager routes all cursor-related commands to the cursor library instead of the driver. The cursor library implements a cursor by fetching the entire result set from the underlying driver and caching the result set on the client. When using the ODBC cursor library, the application is limited to the cursor functionality of the cursor library; any support for additional cursor functionality in the underlying driver is not available to the application.

There is little need to use the ODBC cursor library with the Microsoft® SQL Server[™] ODBC driver because the driver itself supports more cursor functionality than the ODBC cursor library. The only reason to use the ODBC cursor library with the SQL Server ODBC driver is because the driver implements its cursor support through server cursors, and server cursors do not support all SQL statements. Anytime there is a need to have a static cursor with stored procedures, batches, or SQL statements containing COMPUTE, COMPUTE BY, FOR BROWSE, or INTO, consider using the ODBC cursor library. However, care must be used with the cursor library because it caches the entire result set on the client, which can use large amounts of memory and slow performance.

An application invokes the cursor library on a connection-by-connection basis by using **SQLSetConnectAttr** to set the SQL_ATTR_ODBC_CURSORS connection attribute before connecting to a data source. SQL_ATTR_ODBC_CURSORS is set to one of three values:

SQL_CUR_USE_ODBC

When this option is set with the SQL Server ODBC driver, the ODBC cursor library overrides the SQL Server ODBC driver's native cursor support. Only the cursor types supported by the cursor library can be used for the connection; server cursors cannot be used.

SQL_CUR_USE_DRIVER

When this option is set, all of the cursor support native to the SQL Server ODBC driver can be used for the connection. The ODBC cursor library cannot be used. All cursors are implemented as server cursors.

SQL_CUR_USE_IF_NEEDED

When this option is set, the effect is the same as SQL_CUR_USE_DRIVER when used with the SQL Server ODBC driver. At connect time, the ODBC Driver Manager tests to see if the ODBC driver being connected to supports the SQL_FETCH_PRIOR option of **SQLFetchScroll**. If the driver does not support the option, the ODBC Driver Manager loads the ODBC cursor library. If the driver does support the option, the ODBC Driver Manager does not load the ODBC cursor library and the application uses the native support of the driver. Because the SQL Server ODBC driver supports SQL_FETCH_PRIOR, the ODBC Driver Manager does not load the ODBC cursor library.

The cursor library (shipped with the SQL Server ODBC driver) allows applications to use multiple active statements on a connection, as well as scrollable, updatable cursors. The cursor library, Odbccr32.dll for ANSI applications and Odbccu32.dll for Unicode applications, must be loaded to support this functionality. Use **SQLSetConnectAttr** to specify how the cursor library should be used and **SQLSetStmtAttr** to specify the cursor type, concurrency, and rowset size.

See Also

<u>Client Cursors</u> <u>SQLFetchScroll</u> <u>SQLSetConnectAttr</u> <u>SQLSetStmtAttr</u>

Cursor Types

ODBC defines four cursor types supported by Microsoft® SQL Server[™] and the SQL Server ODBC driver. These cursors vary in their ability to detect changes to the result set and in the resources, such as memory and space in **tempdb**, they consume. A cursor can detect changes to rows only when it attempts to refetch those rows; there is no way for the data source to notify the cursor of changes to the currently fetched rows. A cursor's ability to detect changes not made through the cursor is also influenced by the transaction isolation level.

These are the four ODBC cursor types supported by SQL Server:

- Forward-only cursors do not support scrolling; they only support fetching rows serially from the start to the end of the cursor.
- Static cursors are built in **tempdb** when the cursor is opened. They always display the result set as it was when the cursor was opened. They never reflect changes to the data. SQL Server static cursors are always read-only. Because a static server cursor is built as a work table in **tempdb**, the size of the cursor result set cannot exceed the maximum row size allowed by SQL Server.
- Keyset-driven cursors have the membership and order of rows in the result set fixed when the cursor is opened. Changes to nonkey columns are visible through the cursor.
- Dynamic cursors are the opposite of static cursors. Dynamic cursors reflect all changes made to the rows in their result set. The data values, order, and membership of the rows in the result set can change on each fetch.

See Also

Cursor Types

Cursor Behaviors

ODBC supports the SQL-92/ISO options for specifying the behavior of cursors by specifying their scrollability and sensitivity. These behaviors are specified by setting the SQL_ATTR_CURSOR_SCROLLABLE and SQL_ATTR_CURSOR_SENSITIVITY options on a call to **SQLSetStmtAttr**. The Microsoft® SQL Server[™] ODBC driver implements these options by requesting server cursors with the following characteristics:

Cursor behavior settings	Server cursor characteristics requested
SQL_SCROLLABLE and	Keyset-driven cursor and version-based
SQL_SENSITIVE	optimistic concurrency
SQL_SCROLLABLE and	Static cursor and read-only concurrency
SQL_INSENSITIVE	
SQL_SCROLLABLE and	Static cursor and read-only concurrency
SQL_UNSPECIFIED	
SQL_NONSCROLLABLE and	Forward-only cursor and version-based
SQL_SENSITIVE	optimistic concurrency
SQL_NONSCROLLABLE and	Default result set (forward-only, read-only)
SQL_INSENSITIVE	
SQL_NONSCROLLABLE and	Default result set (forward-only, read-only)
SQL_UNSPECIFIED	

Version-based optimistic concurrency requires a **timestamp** column in the underlying table. If version-based optimistic concurrency control is requested on a table that does not have a **timestamp** column, the server uses values-based optimistic concurrency.

Scrollability

When SQL_ATTR_CURSOR_SCROLLABLE is set to SQL_SCROLLABLE, the cursor supports all of the different values for the *FetchOrientation* parameter of **SQLFetchScroll**. When SQL_ATTR_CURSOR_SCROLLABLE is set to SQL_NONSCROLLABLE, the cursor only supports a *FetchOrientation* value of SQL_FETCH_NEXT.

Sensitivity

When SQL_ATTR_CURSOR_SENSITIVITY is set to SQL_SENSITIVE, the cursor reflects data modifications made by the current user or committed by other users. When SQL_ATTR_CURSOR_SENSITIVITY is set to SQL_INSENSITIVE, the cursor does not reflect data modifications.

Cursor Properties

The overall characteristics of a cursor are determined by setting either the ODBC-style cursor type or the SQL-92/ISO cursor behaviors. Additional statement and connection attributes also affect the behavior of a cursor:

- Rowset size statement
- Cursor concurrency statement
- Transaction isolation-level connection

Cursor Rowset Size

ODBC cursors are not limited to fetching one row at a time; they can retrieve multiple rows in each call to **SQLFetch** or **SQLFetchScroll**. When working with a client/server database such as Microsoft® SQL Server[™], it is more efficient to fetch several rows at a time. The number of rows returned on a fetch is called the rowset size and is specified using the

SQL_ATTR_ROW_ARRAY_SIZE of **SQLSetStmtAttr**. Cursors whose rowset size is greater than 1 are called block cursors.

There are two options for binding result set columns for block cursors:

• Column-wise binding

Each column is bound to an array of variables. Each array has the same number of elements as the rowset size.

• Row-wise binding

An array is built using structures that hold the data and indicators for all the columns in a row. The array has the same number of structures as the rowset size.

When either column-wise or row-wise binding is used, each call to **SQLFetch** or **SQLFetchScroll** fills the bound arrays with data from the rowset retrieved.

SQLGetData can also be used to retrieve column data from a block cursor. Because **SQLGetData** works one row at a time, **SQLSetPos** must be called to set a specific row in the rowset as the current row before calling **SQLGetData**.

The SQL Server ODBC driver offers an optimization using rowsets to quickly retrieve an entire result set. To use this optimization, set the cursor attributes to their defaults (forward-only, read-only, rowset size = 1) at the time **SQLExecDirect** or **SQLExecute** is called. The ODBC driver sets up a default result set, which is more efficient than server cursors when only transferring results to the client without scrolling. After the statement has been executed, increase the rowset size and use either column-wise or row-wise binding. This allows SQL Server to use a default result set to efficiently send result rows to the

client, while the ODBC driver continuously pulls rows from the network buffers on the client.

See Also

SQLFetchScroll
SQLGetData
SQLSetStmtAttr

Cursor Concurrency

Cursor operations, like cursor types, are affected by the concurrency options set by the application. Concurrency options are set using the SQL_ATTR_CONCURRENCY option of **SQLSetStmtAttr**. The concurrency types are:

- Read-only (SQL_CONCUR_READONLY)
- Values (SQL_CONCUR_VALUES)
- Row version (SQL_CONCUR_ROWVER)
- Lock (SQL_CONCUR_LOCK)

For more information about the types of locks generated by these concurrency options, see <u>Cursor Concurrency</u>.

See Also

<u>SQLSetStmtAttr</u>

Cursor Transaction Isolation Level

The complete locking behavior of cursors is based on an interaction between concurrency attributes and the transaction isolation level set by the client. ODBC clients set the transaction isolation level using the **SQLSetConnectAttr** SQL_ATTR_TXN_ISOLATION attribute. The locking behavior of a specific cursor environment is determined by combining the locking behaviors of the concurrency and transaction isolation level options.

The following cursor transaction isolation levels are supported by the Microsoft® SQL Server[™] ODBC driver:

- Read committed (SQL_TXN_READ_COMMITTED)
- Read uncommitted (SQL_TXN_READ_UNCOMMITTED)
- Repeatable read (SQL_TXN_REPEATABLE_READ)
- Serializable (SQL_TXN_SERIALIZABLE)

For more information about the types of locks generated by the transaction isolation levels, see <u>Cursor Transaction Isolation Levels</u>.

Note that the ODBC API specifies additional transaction isolation levels, but these are not supported by SQL Server or the SQL Server ODBC driver.

Cursor Programming Details (ODBC)

Choosing the correct cursor type can improve application performance. Under certain conditions, Microsoft® SQL Server[™] may implicitly convert a cursor type if you execute an SQL statement not supported by the cursor type you requested.

See Also

Choosing a Cursor Type

Implicit Cursor Conversions (ODBC)

Applications can request a cursor type through **SQLSetStmtAttr** and then execute an SQL statement that is not supported by server cursors of the type requested. A call to **SQLExecute** or **SQLExecDirect** returns SQL_SUCCESS_WITH_INFO and **SQLGetDiagRec** returns:

```
szSqlState = "01S02", *pfNativeError = 0,
szErrorMsg="[Microsoft][ODBC SQL Server Driver]Cursor type chan
```

The application can determine what type of cursor is now being used by calling **SQLGetStmtOption** with *fOption* set to SQL_CURSOR_TYPE. The cursor type conversion applies to only one statement. The next **SQLExecDirect** or **SQLExecute** will be done using the original statement cursor settings.

See Also

Implicit Cursor Conversions

Using Autofetch with ODBC Cursors

When connected to an instance of Microsoft® SQL Server[™] 2000, the SQL Server ODBC driver supports an autofetch option when using any server cursor type. With autofetch, the **SQLExecute** or **SQLExecDirect** function that opens the cursor also has an implicit **SQLFetchScroll**(SQL_FIRST) function. The rows comprising the first rowset are returned to the bound application variables as part of the statement execution, saving another roundtrip across the network to the server. **SQLGetData** is not supported when the autofetch option is enabled; the result set columns must be bound to program variables.

Applications request autofetch by setting the driver-specific SQL_SOPT_SS_CURSOR_OPTIONS statement attribute to SQL_CO_AF.

Fast Forward-Only Cursors (ODBC)

When connected to an instance of Microsoft® SQL Server[™] 2000, the SQL Server ODBC driver supports performance optimizations for forward-only, read-only cursors. Fast forward-only cursors are implemented internally by the driver and server in a manner very similar to default result sets. Besides having high performance, fast forward-only cursors also have these characteristics:

- **SQLGetData** is not supported. The result set columns must be bound to program variables.
- The server automatically closes the cursor when the end of the cursor is detected. The application must still call **SQLCloseCursor** or **SQLFreeStmt**(SQL_CLOSE), but the driver does not have to send the close request to the server. This saves a roundtrip across the network to the server.

If a result set contains a **text**, **ntext**, or **image** column, a fast forward-only cursor is implicitly converted to a dynamic cursor and SQL_SUCCESS_WITH_INFO is returned to the application. **SQLGetData** is enabled for the dynamic cursor.

The application requests fast forward-only cursors using the driver-specific statement attribute SQL_SOPT_SS_CURSOR_OPTIONS. When set to SQL_CO_FFO, fast forward-only cursors are enabled without autofetch. When set to SQL_CO_FFO_AF, the autofetch option is also enabled. For more information about autofetch, see <u>Using Autofetch with ODBC Cursors</u>.

Fast forward-only cursors with autofetch can be used to retrieve a small result set with only one roundtrip to the server. In these steps, *n* is the number of rows to be returned:

- 1. Set SQL_SOPT_SS_CURSOR_OPTIONS to SQL_CO_FFO_AF.
- 2. Set SQL_ATTR_ROW_ARRAY_SIZE to *n* + 1.

- 3. Bind the result columns to arrays of n + 1 elements (to be safe if n + 1 rows are actually fetched).
- 4. Open the cursor with either **SQLExecDirect** or **SQLExecute**.
- 5. If the return status is SQL_SUCCESS, then call **SQLFreeStmt** or **SQLCloseCursor** to close the cursor. All data for the rows will be in the bound program variables.

With these steps, the **SQLExecDirect** or **SQLExecute** sends a cursor open request with the autofetch option enabled. On that single request from the client, the server:

- Opens the cursor.
- Builds the result set and sends the rows to the client.
- Because the rowset size was set to 1 more than the number of rows in the result set, the server detects the end of the cursor and closes the cursor.

See Also

<u>SQLSetStmtAttr</u>

Scrolling and Fetching Rows

To use a scrollable cursor, an ODBC application must:

- Set the cursor capabilities using **SQLSetStmtAttr**.
- Open the cursor using **SQLExecute** or **SQLExecDirect**.
- Scroll and fetch rows using **SQLFetch** or **SQLFetchScroll**.

Both **SQLFetch** and **SQLFetchSroll** can fetch blocks of rows at a time. The number of rows returned is specified using **SQLSetStmtAttr** to set the SQL_ATTR_ROW_ARRAY_SIZE parameter.

ODBC applications can use **SQLFetch** to fetch through a forward-only cursor.

SQLFetchScroll is used to scroll around a cursor. **SQLFetchScroll** supports fetching the next, prior, first, and last rowsets, as well as relative fetching (fetch the rowset *n* rows from the start of the current rowset) and absolute fetching (fetch the rowset starting at row *n*). If *n* is negative in an absolute fetch, rows are counted from the end of the result set. Thus, an absolute fetch of row -1 means to fetch the rowset that starts with the last row in the result set.

Applications that use **SQLFetchScroll** only for its block cursor capabilities, such as reports, are likely to pass through the result set a single time, using only the option to fetch the next rowset. Screen-based applications, on the other hand, can take advantage of all of the capabilities of **SQLFetchScroll**. If the application sets the rowset size to the number of rows displayed on the screen and binds the screen buffers to the result set, it can translate scroll bar operations directly to calls to **SQLFetchScroll**.

Scroll bar operation	SQLFetchScroll scrolling option
Page up	SQL_FETCH_PRIOR
Page down	SQL_FETCH_NEXT
Line up	SQL_FETCH_RELATIVE with FetchOffset
	equal to -1

Line down	SQL_FETCH_RELATIVE with FetchOffset
	equal to 1
Scroll box to top	SQL_FETCH_FIRST
Scroll box to bottom	SQL_FETCH_LAST
Random scroll box position	SQL_FETCH_ABSOLUTE

To fetch and update rowsets

Bookmarking Rows

A bookmark is a value used to identify a row of data. The meaning of the bookmark value is known only to the driver or data source. For example, it might be as simple as a row number or as complex as a disk address. In ODBC, the application requests a bookmark for a particular row, stores it, and passes it back to the cursor to return to the row.

When fetching rows with **SQLFetchScroll**, an application can use a bookmark as a basis for selecting the starting row. This is a form of absolute addressing because it does not depend on the current cursor position. To scroll to a bookmarked row, the application calls **SQLFetchScroll** with a FetchOrientation of SQL_FETCH_BOOKMARK. This operation uses the bookmark pointed to by the SQL_ATTR_FETCH_BOOKMARK_PTR option attribute. It returns the rowset starting with the row identified by that bookmark. An application can specify an offset for this operation in the FetchOffset argument of the call to **SQLFetchScroll**. When an offset is specified, the first row of the returned rowset is determined by adding the number in the FetchOffset argument to the number of the row identified by the bookmark. The Microsoft® SQL ServerTM ODBC driver only supports bookmarks on static and keyset cursors. If a dynamic cursor is requested when bookmarks are set on, a keyset cursor is opened instead.

Bookmarks can also be used with **SQLBulkOperations** to perform operations on a set of rows starting at the bookmark.

To fetch and update rowsets

Positioned Updates (ODBC)

ODBC supports two methods for performing positioned updates in a cursor:

- SQLSetPos
- WHERE CURRENT OF clause

The most common approach is to use **SQLSetPos**, which has the following options:

SQL_POSITION

Positions the cursor on a specific row in the current rowset.

SQL_REFRESH

Refreshes program variables bound to the result set columns with the values from the row the cursor is currently positioned on.

SQL_UPDATE

Updates the current row in the cursor with the values stored in the program variables bound to the result set columns.

SQL_DELETE

Deletes the current row in the cursor.

SQLSetPos can be used with any statement result set when the statement handle cursor attributes are set to use server cursors. The result set columns must be bound to program variables. Once the application has fetched a row it calls **SQLSetPos**(SQL_POSTION) to position the cursor on the row. The application could then call SQLSetPos(SQL_DELETE) to delete the current row, or it can move new data values into the bound program variables and call SQLSetPos(SQL_UPDATE) to update the current row.

Applications can update or delete any row in the rowset with **SQLSetPos**. Calling **SQLSetPos** is a convenient alternative to constructing and executing an SQL statement. **SQLSetPos** operates on the current rowset and can be used only after a call to **SQLFetchScroll**. Rowset size is set by a call to **SQLSetStmtAttr** with an attribute argument of SQL_ATTR_ROW_ARRAY_SIZE. **SQLSetPos** uses a new rowset size, however, only after a call to **SQLFetch** or **SQLFetchScroll**. For example, if the rowset size is changed, then **SQLSetPos** is called, and then **SQLFetch** or **SQLFetchScroll** is called; the call to **SQLSetPos** uses the old rowset size, but **SQLFetch** or **SQLFetchScroll** uses the new rowset size.

The first row in the rowset is row number 1. The RowNumber argument in **SQLSetPos** must identify a row in the rowset; that is, its value must be in the range between 1 and the number of rows that were most recently fetched (which may be less than the rowset size). If RowNumber is 0, the operation applies to every row in the rowset.

The delete operation of **SQLSetPos** makes the data source delete one or more selected rows of a table. To delete rows with **SQLSetPos**, the application calls **SQLSetPos** with Operation set to SQL_DELETE and RowNumber set to the number of the row to delete. If RowNumber is 0, all rows in the rowset are deleted.

After **SQLSetPos** returns, the deleted row is the current row, and its status is SQL_ROW_DELETED. The row cannot be used in any further positioned operations, such as calls to **SQLGetData** or **SQLSetPos**.

When deleting all rows of the rowset (RowNumber is equal to 0), the application can prevent the driver from deleting certain rows by using the row operation array in the same way as for the update operation of **SQLSetPos**.

Every row that is deleted should be a row that exists in the result set. If the application buffers were filled by fetching and if a row status array has been maintained, its values at each of these row positions should not be SQL_ROW_DELETED, SQL_ROW_ERROR, or SQL_ROW_NOROW.

Positioned updates can also be done using the WHERE CURRENT OF clause on UPDATE, DELETE, and INSERT statements. WHERE CURRENT OF requires a cursor name, which ODBC will generate when the **SQLGetCursorName** function is called, or which you can specify by calling **SQLSetCursorName**. The general steps to perform a WHERE CURRENT OF update in an ODBC application are:

• Call **SQLSetCursorName** to establish a cursor name for the statement handle.

- Build a SELECT statement with a FOR UPDATE OF clause and execute it.
- Call **SQLFetchScroll** to retrieve a rowset or **SQLFetch** to retrieve a row.
- Call **SQLSetPos** (SQL_POSITION) to position the cursor on the row.
- Build and execute an UPDATE statement with a WHERE CURRENT OF clause using the cursor name set with **SQLSetCursorName**.

As an alternative, you could call **SQLGetCursorName** after executing the SELECT statement instead of calling **SQLSetCursorName** before executing the SELECT statement. **SQLGetCursorName** returns a default cursor name assigned by ODBC if you do not set a cursor name using **SQLSetCursorName**.

SQLSetPos is preferred over WHERE CURRENT OF when using server cursors. If you are using a static, updatable cursor with the ODBC cursor library, the cursor library implements WHERE CURRENT OF updates by adding a WHERE clause with the key values for the underlying table. This can cause unintended updates if the keys in the table are not unique.

To fetch and update rowsets

Performing Transactions

Microsoft® SQL Server[™] and the SQL Server ODBC driver support the ODBC API transaction management functions. Microsoft offers full support for local transactions on an individual server. The SQL Server ODBC driver uses these features to support the ODBC API functions that manage transactions.

Through the use of the Microsoft Distributed Transaction Coordinator (MS DTC), the SQL Server ODBC driver can participate in distributed transactions spanning multiple servers.

See Also

Transactions

Transactions in ODBC

Transactions in ODBC are managed at the connection level. When an application completes a transaction, it commits or rolls back all work done through all statement handles on that connection. To commit or roll back a transaction, applications should call **SQLEndTran** rather than submitting a COMMIT or ROLLBACK statement.

An application calls **SQLSetConnectAttr** to switch between the two ODBC modes of managing transactions:

• Autocommit mode

Each individual statement is automatically committed when it completes successfully. When running in autocommit mode no other transaction management functions are needed.

• Manual-commit mode

All executed statements are included in the same transaction until it is specifically terminated by calling **SQLEndTran**.

Autocommit mode is the default transaction mode for ODBC. When a connection is made, it is in autocommit mode until **SQLSetConnectAttr** is called to switch to manual-commit mode by setting autocommit mode off. When an application turns autocommit off, the next statement sent to the database starts a transaction. The transaction then remains in effect until the application calls **SQLEndTran** with either the SQL_COMMIT or SQL_ROLLBACK options. The command sent to the database after **SQLEndTran** starts the next transaction.

If an application switches from manual-commit to autocommit mode, the driver commits any transactions currently open on the connection.

ODBC applications should not use Transact-SQL transaction statements (such as BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION) because this can result in indeterminate behavior in the driver. An ODBC application should either:

• Run in autocommit mode and not use any transaction management

functions or statements.

-or-

• Run in manual-commit mode and use the ODBC **SQLEndTran** function to either commit or roll back transactions.

See Also

SQLEndTran

<u>SQLSetConnectAttr</u>

Performing Distributed Transactions

The Microsoft Distributed Transaction Coordinator (MS DTC) allows applications to extend transactions across two or more instances of Microsoft® SQL ServerTM. It also allows applications to participate in transactions managed by transaction managers that comply with the X/Open DTP XA standard. ODBC applications that use SQL Server version 6.5 or later can participate in MS DTC transactions.

Normally, all transaction management commands are sent through the ODBC driver to the server. The application starts a transaction by calling **SQLSetConnectAttr** with the autocommit mode turned off. The application then performs the updates comprising the transaction and calls **SQLEndTran** with either the SQL_COMMIT or SQL_ROLLBACK option.

When using MS DTC, however, MS DTC becomes the transaction manager and the application no longer uses **SQLEndTran**.

To use Microsoft Distributed Transaction Coordinator

Handling Errors and Messages

When an application calls an ODBC function, the driver executes the function and returns diagnostic information in two ways: A return code indicates the overall success or failure of an ODBC function and diagnostic records provide detailed information about the function. Diagnostic records include a header record and status records. At least one diagnostic record, the header record, is returned even if the function succeeds.

Diagnostic information is used at development time to catch programming errors, such as invalid handles and syntax errors in hard-coded SQL statements. It is also used at run time to catch run-time errors and warnings, such as data truncation, rule violations, and syntax errors in SQL statements entered by the user. Program logic is generally based on return codes.

For example, after an application calls **SQLFetch** to retrieve the rows in a result set, the return code indicates if the end of the result set was reached (SQL_NO_DATA), if any informational messages were returned (SQL_SUCCESS_WITH_INFO), or if an error occurred (SQL_ERROR).

If an ODBC driver returns anything other than SQL_SUCCESS, then the application can call **SQLGetDiagRec** to retrieve any informational or error messages present. Use **SQLGetDiagRec** to scroll up and down the message set if there is more than one message.

The return code SQL_INVALID_HANDLE always indicates a programming error and should never be encountered at run time. All other return codes provide run-time information, although SQL_ERROR may indicate a programming error.

The original Microsoft® SQL Server[™] native API, DB-Library for C, allows an application to install callback error-handling and message-handling functions that return errors or messages. Some Transact-SQL statements, such as PRINT, RAISERROR, DBCC, and SET, return their results to the DB-Library message handler function instead of to a result set. However, the ODBC API has no such callback capability, so when the SQL Server ODBC driver detects messages coming back from SQL Server, it sets the ODBC return code to SQL_SUCCESS_WITH_INFO or SQL_ERROR and returns the message as one

or more diagnostic records. Therefore, an ODBC application must carefully test for these return codes and call **SQLGetDiagRec** to retrieve message data.

To process ODBC errors

Processing Statements That Generate Messages

Using SET SHOWPLAN and SET STATISTICS

The Transact-SQL SET statement options STATISTICS TIME and STATISTICS IO are used to get information that aids in diagnosing long-running queries. Earlier versions of Microsoft® SQL Server[™] also support the SHOWPLAN option for analyzing query plans. An ODBC application can set these options by executing the following statements:

SQLExecDirect(hstmt, "SET SHOWPLAN ON", SQL_NTS); SQLExecDirect(hstmt, "SET STATISTICS TIME ON", SQL_NTS); SQLExecDirect(hstmt, "SET STATISTICS IO ON", SQL_NTS);

When SET STATISTICS TIME or SET SHOWPLAN are ON, **SQLExecute** and **SQLExecDirect** return SQL_SUCCESS_WITH_INFO, and, at that point, the application can retrieve the SHOWPLAN or STATISTICS TIME output by calling **SQLGetDiagRec** until it returns SQL_NO_DATA. Each line of SHOWPLAN data comes back in the format:

szSqlState="01000", *pfNativeError=6223, szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server] Table Scan"

SQL Server version 7.0 replaces the SHOWPLAN option with SHOWPLAN_ALL and SHOWPLAN_TEXT, both of which return output as a result set, not a set of messages.

Each line of STATISTICS TIME comes back in the format:

szSqlState="01000", *pfNativeError= 3613, szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server] SQL Server Parse and Compile Time: cpu time = 0 ms."

The output of SET STATISTICS IO is not available until the end of a result set. To get STATISTICS IO output, the application calls **SQLGetDiagRec** at the

time **SQLFetch** or **SQLFetchScroll** returns SQL_NO_DATA. The output of STATISTICS IO comes back in the format:

```
szSqlState="01000", *pfNativeError= 3615,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
Table: testshow scan count 1, logical reads: 1,
physical reads: 0."
```

Using DBCC Statements

DBCC statements return their data as messages, not result sets. **SQLExecDirect** or **SQLExecute** return SQL_SUCCESS_WITH_INFO, and the application retrieves the output by calling **SQLGetDiagRec** until it returns SQL_NO_DATA.

For example, the following statement returns SQL_SUCCESS_WITH_INFO:

SQLExecDirect(hstmt, "DBCC CHECKTABLE(authors)", SQL_NTS)

Calls to **SQLGetDiagRec** return:

```
szSqlState = "01000", *pfNativeError = 2536,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
Checking authors"
szSqlState = "01000", *pfNativeError = 2579,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
The total number of data pages in this table is 1."
szSqlState = "01000", *pfNativeError = 7929,
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
Table has 23 data rows."
szSqlState = "01000", *pfNativeError = 2528
szErrorMsg="[Microsoft][ODBC SQL Server Driver][SQL Server]
DBCC execution completed. If DBCC printed error messages,
see your System Administrator."
```

Using PRINT and RAISERROR Statements

Transact-SQL PRINT and RAISERROR statements also return data by calling **SQLGetDiagRec**. PRINT statements cause the SQL statement execution to return SQL_SUCCESS_WITH_INFO, and a subsequent call to **SQLGetDiagRec** returns a *SQLState* of 01000. A RAISERROR with a severity of ten or lower behaves the same as PRINT. A RAISERROR with a severity of 11 or higher causes the execute to return SQL_ERROR, and a subsequent call to **SQLGetDiagRec** returns *SQLState* 42000. For example, the following statement returns SQL_SUCCESS_WITH_INFO:

SQLExecDirect (hstmt, "PRINT 'Some message' ", SQL_NTS);

Calling **SQLGetDiagRec** returns:

```
szSQLState = "01000", *pfNative Error = 0,
szErrorMsg= "[Microsoft] [ODBC SQL Server Driver][SQL Server]
Some message"
```

The following statement returns SQL_SUCCESS_WITH_INFO:

SQLExecDirect (hstmt, "RAISERROR ('Sample error 1.', 10, -1)", SQL_NTS)

Calling **SQLGetDiagRec** returns:

```
szSQLState = "01000", *pfNative Error = 50000,
szErrorMsg= "[Microsoft] [ODBC SQL Server Driver][SQL Server]
Sample error 1."
```

The following statement returns SQL_ERROR:

SQLExecDirect (hstmt, "RAISERROR ('Sample error 2.', 11, -1)", SQ]

Calling **SQLGetDiagRec** returns:

```
szSQLState = "42000", *pfNative Error = 50000,
szErrorMsg= "[Microsoft] [ODBC SQL Server Driver][SQL Server]
Sample error 2."
```

The timing of calling **SQLGetDiagRec** is critical when output from PRINT or RAISERROR statements is included in a result set. The call to **SQLGetDiagRec** to retrieve the PRINT or RAISERROR output must be made immediately after the statement that receives SQL_ERROR or SQL_SUCCESS_WITH_INFO. This is straightforward when only a single SQL statement is executed, as in the examples above. In these cases, the call to **SQLExecDirect** or **SQLExecute** returns SQL_ERROR or SQL_SUCCESS_WITH_INFO and **SQLGetDiagRec** can then be called. It is less straightforward when coding loops to handle the output of a batch of SQL statements or when executing SQL Server stored procedures.

In this case, SQL Server returns a result set for every SELECT statement executed in a batch or stored procedure. If the batch or procedure contains PRINT or RAISERROR statements, the output for these is interleaved with the SELECT statement result sets. If the first statement in the batch or procedure is a PRINT or RAISERROR, the **SQLExecute** or **SQLExecDirect** returns SQL_SUCCESS_WITH_INFO or SQL_ERROR, and the application needs to call **SQLGetDiagRec** until it returns SQL_NO_DATA to retrieve the PRINT or RAISERROR information.

If the PRINT or RAISERROR statement comes after an SQL statement (such as a SELECT statement), then the PRINT or RAISERROR information is returned when **SQLMoreResults** positions on the result set containing the error. **SQLMoreResults** returns SQL_SUCCESS_WITH_INFO or SQL_ERROR depending on the severity of the message. Messages are retrieved by calling **SQLGetDiagRec** until it returns SQL_NO_DATA.

See Also

SQLMoreResults

Diagnostic Records and Fields

Diagnostic records are associated with ODBC environment, connection, statement, or descriptor handles. When any ODBC function raises a return code other than SQL_SUCCESS or SQL_INVALID_HANDLE, the handle called by the function has associated diagnostic records that contain informational or error messages. These records are retained until another function is called using that handle, at which time they are discarded. There is no limit to the number of diagnostic records that can be associated with a handle at any one time.

There are two types of diagnostic records: header and status. The header record is record 0; when there are status records, they are records 1 and later. Diagnostic records contain different fields for the header record and the status records. ODBC components can also define their own diagnostic record fields.

Fields in the header record contain general information about a function's execution, including the return code, row count, number of status records, and type of statement executed. The header record is always created unless an ODBC function returns SQL_INVALID_HANDLE. For a complete list of fields in the header record, see **SQLGetDiagField**.

Fields in the status records contain information about specific errors or warnings returned by the ODBC Driver Manager, driver, or data source, including the SQLSTATE, native error number, diagnostic message, column number, and row number. Status records are created only if the function returns SQL_ERROR, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_NEED_DATA, or SQL_STILL_EXECUTING. For a complete list of fields in the status records, see **SQLGetDiagField**.

SQLGetDiagRec retrieves a single diagnostic record along with its ODBC SQLSTATE, native error number, and diagnostic-message fields. This functionality is similar to the ODBC 2.*x* **SQLError** function. The simplest error-handling function in ODBC 3.*x* is to repeatedly call **SQLGetDiagRec** starting with the *RecNumber* parameter set to 1 and incrementing *RecNumber* by 1 until **SQLGetDiagRec** returns SQL_NO_DATA. This is equivalent to an ODBC 2.*x* application calling **SQLError** until it returns SQL_NO_DATA_FOUND.

ODBC 3.x supports much more diagnostic information than ODBC 2.x. This

information is stored in additional fields in diagnostic records retrieved by using **SQLGetDiagField**.

The Microsoft® SQL Server[™] ODBC driver has driver-specific diagnostic fields that can be retrieved with **SQLGetDiagField**. Labels for these driver-specific fields are defined in Odbcss.h. Use these labels to retrieve the SQL Server state, severity level, server name, procedure name, and line number associated with each diagnostic record. Also, Odbcss.h contains definitions of the codes the driver uses to identify Transact-SQL statements if an application calls **SQLGetDiagField** with *DiagIdentifier* set to SQL_DIAG_DYNAMIC_FUNCTION_CODE.

SQLGetDiagField is processed by the ODBC Driver Manager using error information it caches from the underlying driver. The ODBC Driver Manager does not cache driver-specific diagnostic fields until after a successful connection has been made. **SQLGetDiagField** returns SQL_ERROR if it is called to get driver-specific diagnostic fields before a successful connection has been completed. If an ODBC connect function returns SQL_SUCCESS_WITH_INFO, the driver-specific diagnostic fields for the connect function are not yet available. You can start calling **SQLGetDiagField** for driver-specific diagnostic fields only after you have made another ODBC function call after the connect function.

Most errors reported by the SQL Server ODBC driver can be effectively diagnosed using only the information returned by **SQLGetDiagRec**. In some cases, however, the information returned by the driver-specific diagnostic fields is important in diagnosing an error. When coding an ODBC error handler for applications using the SQL Server ODBC driver, it is a good idea to also use **SQLGetDiagField** to retrieve at least the SQL_DIAG_SS_MSGSTATE and SQL_DIAG_SS_SEVERITY driver-specific fields. If a particular error can be raised at several locations in the SQL Server code, SQL_DIAG_SS_MSGSTATE indicates to a Microsoft support engineer specifically where an error was raised, which sometimes aids in diagnosing a problem.

To process ODBC errors

Native Error Numbers

For errors that occur in the data source (returned by Microsoft® SQL Server[™]), the SQL Server ODBC driver returns the native error number returned to it by SQL Server. For errors detected by the driver, the SQL Server driver returns a native error number of 0. For more information about a list of native error numbers, see the error column of the **sysmessages** system table in the **master** database in SQL Server.

For errors returned by the Net-Library, the native error number is from the underlying network software.

For errors returned by Microsoft Windows NT® 4.0 or Microsoft Windows® 95, the SQL Server ODBC driver calls the Microsoft Win32® **GetLastError** function and returns that error as the native error.

To process ODBC errors

SQLSTATE (ODBC Error Codes)

SQLSTATEs provide detailed information about the cause of a warning or error. For errors that occur in the data source, detected and returned by Microsoft® SQL Server[™], the SQL Server ODBC driver maps the returned native error number to the appropriate SQLSTATE. If a native error number does not have an ODBC error code to map to, the SQL Server ODBC driver returns SQLSTATE 42000 ("syntax error or access violation"). For errors that are detected by the driver, the SQL Server ODBC driver generates the appropriate SQLSTATE.

To process ODBC errors

Error Messages

The text of messages returned by the Microsoft® SQL ServerTM ODBC driver is placed in the *MessageText* parameter of **SQLGetDiagRec**. The source of an error is indicated by the header of the message:

[Microsoft][ODBC Driver Manager]

These errors are raised by the ODBC Driver Manager.

[Microsoft][ODBC Cursor Library]

These errors are raised by the ODBC cursor library.

[Microsoft][ODBC SQL Server Driver]

These errors are raised by the SQL Server ODBC driver. If there are no other nodes with either the name of a Net-Library or SQL Server, then the error was encountered in the driver.

[Microsoft][ODBC SQL Server Driver][*Net-Libraryname*]

These errors are raised by the SQL Server Net-Library, where *Net-Libraryname* is the display name of a SQL Server client Net-Library (for example, Named Pipes, Shared Memory, Multiprotocol, TCP/IP Sockets, NWLink IPX/SPX, or Banyan VINES). The remainder of the error message contains the Net-Library function called and the function called in the underlying network API by the TDS function. The *pfNative* error code returned with these errors is the error code from the underlying network protocol stack.

[Microsoft][ODBC SQL Server Driver][SQL Server]

These errors are raised by SQL Server. The remainder of the error message is the text of the error message from SQL Server. The *pfNative* code returned with these errors is the error number from SQL Server. For more information about a list of error messages (and their numbers) that can be returned by SQL Server, see the description and error columns of the **sysmessages** system table in the **master** database in SQL Server.

To process ODBC errors

Running Stored Procedures

A stored procedure is an executable object stored in a database. Microsoft® SQL ServerTM supports:

• Stored procedures

One or more SQL statements precompiled into a single executable procedure.

• Extended stored procedures

C or C++ dynamic-link libraries (DLL) written to the SQL Server Open Data Services API for extended stored procedures. The Open Data Services API extends the capabilities of stored procedures to include C or C++ code.

When executing statements, calling a stored procedure on the data source (instead of directly executing or preparing a statement in the client application) can provide:

• Higher performance

SQL statements are parsed and compiled when procedures are created. This overhead is then saved when the procedures are executed.

• Reduced network overhead

Executing a procedure instead of sending complex queries across the network can reduce network traffic. If an ODBC application uses the ODBC { CALL } syntax to execute a stored procedure, the ODBC driver makes additional optimizations that eliminate the need to convert parameter data.

• Greater consistency

If an organization's rules are implemented in a central resource, such as a stored procedure, they can be coded, tested, and debugged once. Individual programmers can then use the tested stored procedures instead of developing their own implementations. • Greater accuracy

Because stored procedures are usually developed by experienced programmers, they tend to be more efficient and have fewer errors than code developed multiple times by programmers of varying skill levels.

• Added functionality

Extended stored procedures can use C and C++ features not available in Transact-SQL statements.

To call remote procedures

Calling a Stored Procedure

The Microsoft® SQL Server[™] ODBC driver supports both the ODBC CALL escape sequence and the Transact-SQL EXECUTE statement for executing stored procedures; the ODBC CALL escape sequence is the preferred method. Using ODBC syntax enables an application to retrieve the return codes of stored procedures and the SQL Server ODBC driver is also optimized to use a protocol originally developed for sending remote procedure (RPC) calls between SQL Servers. This RPC protocol increases performance by eliminating much of the parameter processing and statement parsing done on the server.

The ODBC CALL escape sequence for calling a procedure is:

{[?=]call procedure_name[([parameter][,[parameter]]...)]}

where *procedure_name* specifies the name of a procedure and *parameter* specifies a procedure parameter.

A procedure can have zero or more parameters. It can also return a value (as indicated by the optional parameter marker ?= at the start of the syntax). If a parameter is an input or an input/output parameter, it can be a literal or a parameter marker. If the parameter is an output parameter, it must be a parameter marker because the output is unknown. Parameter markers must be bound with **SQLBindParameter** before the procedure call statement is executed.

Input and input/output parameters can be omitted from procedure calls. If a procedure is called with parentheses but without any parameters, the driver instructs the data source to use the default value for the first parameter. For example:

{call procedure_name()}

If the procedure does not have any parameters, the procedure can fail. If a procedure is called without parentheses, the driver does not send any parameter values. For example:

{**call** *procedure_name*}

Literals can be specified for input and input/output parameters in procedure calls. For example, the procedure InsertOrder has five input parameters. The

following call to InsertOrder omits the first parameter, provides a literal for the second parameter, and uses a parameter marker for the third, fourth, and fifth parameters. (Parameters are numbered ordinally, beginning with a value of 1.)

```
{call InsertOrder(, 10, ?, ?, ?)}
```

Note that if a parameter is omitted, the comma delimiting it from other parameters must still appear. If an input or input/output parameter is omitted, the procedure uses the default value of the parameter. Other ways to specify the default value of an input or input/output parameter are to set the value of the length/indicator buffer bound to the parameter to SQL_DEFAULT_PARAM, or to use the DEFAULT keyword.

If an input/output parameter is omitted, or if a literal is supplied for the parameter, the driver discards the output value. Similarly, if the parameter marker for the return value of a procedure is omitted, the driver discards the return value. Finally, if an application specifies a return value parameter for a procedure that does not return a value, the driver sets the value of the length/indicator buffer bound to the parameter to SQL_NULL_DATA.

Delimiters in CALL statements

The Microsoft SQL Server ODBC driver by default also supports a compatibility option specific to the ODBC { CALL } escape sequence. The driver will accept CALL statements with only a single set of double quotation marks delimiting the entire stored procedure name:

```
{ CALL "master.dbo.sp_who" }
```

By default the SQL Server ODBC driver also accepts CALL statements that follow the SQL-92 rules and enclose each identifier in double quotation marks:

```
{ CALL "master"."dbo"."sp_who" }
```

When running with the default settings, however, the SQL Server ODBC driver does not support using either form of quoted identifier with identifiers that contain characters not specified as legal in identifiers by the SQL-92 standard. For example, the driver cannot access a stored procedure named "**My.Proc**" using a CALL statement with quoted identifiers:

{ CALL "MyDB"."MyOwner"."My.Proc" }

This statement is interpreted by the driver as:

{ CALL MyDB.MyOwner.My.Proc }

The server will raise an error that a linked server named **MyDB** does not exist.

The issue does not exist when using bracketed identifiers, this statement is interpreted correctly:

{ CALL [MyDB].[MyOwner].[My.Table] }

Users needing to access objects with periods in their identifiers can also use the **odbccmpt** command prompt utility to allow this. The **odbccmpt** utility supports a /Q switch to enforce ODBC and SQL-92 compliant behaviors of quoted identifiers on the CALL statement. To turn the standard compliant behavior on for an application, use the following code, where **file_name** is the name of the application executable file without the path or .exe extension.

```
odbccmpt file_name /Q
```

For more information, see <u>odbccmpt Utility</u>.

To turn the standard compliance off for an application, use the following code.

```
odbccmpt file_name /Q /d
```

Running **odbccmpt** with only the /Q switch adds this key to the Windows NT registry, running with both /Q and /d deletes the key.

HKEY_LOCAL_MACHINE

SOFTWARE

Microsoft

MSSQLServer

Client

ODBCQIBehavior

file_name:REG_SZ:NEW

When the standard compliance option is on for an application, CALL statements cannot use just a single set of double quotation marks around the complete, qualified procedure name. Only individual identifiers can be quoted. Some examples of valid statements are:

```
{ CALL "MyDB"."MyUserID"."My.Proc" }
{ CALL "MyDB".MyUserID."My.Proc" }
{ CALL MyDB.MyUserID."My.Proc" }
```

When the standard compliance option is on, the SQL Server ODBC driver supports using quoted identifiers that contain characters not allowed in SQL-92 identifiers.

To call remote procedures

Batching Stored Procedure Calls

The Microsoft® SQL Server[™] ODBC driver automatically batches stored procedure calls to the server when appropriate. The driver only does this when the ODBC CALL escape sequence is used; it does not do this for the Transact-SQL EXECUTE statement. Batching stored procedure calls can reduce the number of roundtrips to the server and significantly increase performance.

The driver batches procedure calls to the server when you execute a batch containing multiple ODBC CALL escape sequences. It also batches procedure calls when bound parameter arrays are used with an ODBC CALL escape sequence. For example, if you use either row-wise or column-wise parameter binding to bind an array with five elements to the parameters of an ODBC CALL SQL statement, then when **SQLExecute** or **SQLExecDirect** is called, the driver sends a single batch with five procedure calls to the server.

To call remote procedures

Processing Stored Procedure Results

Microsoft[®] SQL Server[™] stored procedures have four mechanisms used to return data:

- Each SELECT statement in the procedure generates a result set.
- The procedure can return data through output parameters.
- A cursor output parameter can pass back a Transact-SQL server cursor.
- The procedure can have an integer return code.

Applications must be able to handle all of these outputs from stored procedures. The CALL or EXECUTE statement should include parameter markers for the return code and output parameters. Use **SQLBindParameter** to bind them all as output parameters and the ODBC driver will transfer the output values to the bound variables. Output parameters and return codes are the last items returned to the client by SQL Server; they are not returned to the application until **SQLMoreResults** returns SQL_NO_DATA.

ODBC does not support binding Transact-SQL cursor parameters. Since all output parameters must be bound before executing a procedure, any Transact-SQL stored procedure that contains an output cursor parameter cannot be called by ODBC applications.

To call remote procedures

Using Catalog Functions

All databases have a structure containing the data stored in the database. A definition of this structure, along with other information such as permissions, is stored in a catalog (implemented as a set of system tables), also known as a data dictionary.

The Microsoft® SQL Server[™] ODBC driver enables an application to determine the database structure through calls to ODBC catalog functions. Catalog functions return information in result sets and are implemented using catalog stored procedures to query the system tables in the catalog. For example, an application might request a result set containing information about all the tables on the system or all the columns in a particular table. The standard ODBC catalog functions are used to get catalog information from the SQL Server to which the application connected.

SQL Server supports distributed queries in which data from multiple, heterogeneous OLE DB data sources is accessed in a single query. One of the methods of accessing a remote OLE DB data source is to define the data source as a linked server. This can be done by using **sp_addlinkserver**. After the linked server has been defined, objects in that server can be referenced in Transact-SQL statements by using a four part name:

linked_server_name.catalog.schema.object_name

The SQL Server ODBC driver supports two driver-specific functions that help get catalog information from linked servers:

• SQLLinkedServers

Returns a list of the linked servers defined to the local server.

• SQLLinkedCatalogs

Returns a list of the catalogs contained in a linked server.

After you have a linked server name and a catalog name, the SQL Server ODBC driver supports getting information from the catalog by using a two part name of *linked_server_name.catalog* for *CatalogName* on the following ODBC catalog functions:

SQLColumnPrivileges	SQLColumns	SQLPrimaryKeys
SQLStatistics	SQLTablePrivileges	SQLTables

The two part *linked_server_name.catalog* is also supported for *FKCatalogName* and *PKCatalogName* on **SQLForeignKeys**.

Using **SQLLinkedServers** and **SQLLinkedCatalogs** requires the following files:

• Odbcss.h

Includes function prototypes and constant definitions for the linked server catalog functions. Odbcss.h must be included in the ODBC application and must be in the include path when the application is compiled.

• Odbcbcp.lib

Must be in the library path of the linker and specified as a file to be linked. Odbcbcp.lib is distributed with the SQL Server ODBC driver.

• Odbcbcp.dll

Must be present at execution time. Odbcbcp.dll is distributed with the SQL Server ODBC driver.

See Also

Distributed Queries sp_addlinkedserver SQLColumnPrivileges SQLColumns SQLForeignKeys SQLLinkedCatalogs SQLLinkedServers SQLPrimaryKeys <u>SQLTablePrivileges</u>

SQLTables

SQLStatistics

Performing Bulk Copy Operations

The Microsoft® SQL Server[™] bulk copy feature supports the transfer of large amounts of data into or out of a SQL Server table or view. Data can also be transferred out by specifying a SELECT statement. The data can be moved between SQL Server and an operating-system data file, such as an ASCII file. The data file can have different formats; the format is defined to bulk copy in a format file. Optionally, data can be loaded into program variables and transferred to SQL Server using bulk copy functions. This is typically much faster than using INSERT statements or calling **SQLBulkOperations** with SQL_ADD.

The ODBC standard does not directly support SQL Server bulk copy operations. When connected to an instance of SQL Server version 6.0 or later, the SQL Server 2000 ODBC driver supports the DB-Library functions that perform SQL Server bulk copy operations. This driver-specific extension provides an easy upgrade path for existing DB-Library applications that use bulk copy functions. The specialized bulk copy support is in the following files:

• Odbcss.h

Includes function prototypes and constant definitions for bulk copy functions. Odbcss.h must be included in the ODBC application performing bulk copy operations and must be in the application's include path when it is compiled.

• Odbcbcp.lib

Must be in the library path of the linker and specified as a file to be linked. Odbcbcp.lib is distributed with the SQL Server ODBC driver.

• Odbcbcp.dll

Must be present at execution time. Odbcbcp.dll is distributed with the SQL Server ODBC driver.

An application typically uses bulk copy in one of the following ways:

• Bulk copy from a table, view, or the result set of a Transact-SQL statement into a data file where the data is stored in the same format as the table or view.

This is called a native-mode data file.

• Bulk copy from a table, view, or the result set of a Transact-SQL statement into a data file where the data is stored in a format other than the one of the table or view.

In this case, a separate format file is created that defines the characteristics (data type, position, length, terminator, and so on) of each column as it is stored in the data file. If all columns are converted to character format, the resulting file is called a character-mode data file.

• Bulk copy from a data file into a table or view.

If needed, a format file is used to determine the layout of the data file.

• Load data into program variables, then import the data into a table or view using the bulk copy functions for bulk copying in a row at a time.

Data files used by bulk copy functions do not have to be created by another bulk copy program. Any other system can generate a data file and format file according to bulk copy definitions; these files can then be used with a SQL Server bulk copy program to import data into SQL Server. For example, you could export data from a spreadsheet in a tab-delimited file, build a format file describing the tab-delimited file, and then use a bulk copy program to quickly import the data into SQL Server. Data files generated by bulk copy can also be imported into other applications. For example, you could use bulk copy functions to export data from a table or view into a tab-delimited file that could then be loaded into a spreadsheet.

Programmers coding applications to use the bulk copy functions should follow the general rules for good bulk copy performance. For more information, see Factors Affecting Bulk Copy Performance.

Note The ODBC **SQLBulkOperations** function has no relationship to the SQL Server bulk copy functions. Applications must use the SQL Server-specific bulk-copy functions to perform bulk copy operations.

Logged and Nonlogged Bulk Copies

Microsoft® SQL Server[™] bulk copies that import data into an instance of SQL Server are run in either logged or nonlogged mode. The difference between logged and nonlogged bulk copy operations is how much information is logged. Both logged and nonlogged bulk copy operations can be rolled back, but only a logged bulk copy operation can be rolled forward.

In a logged bulk copy all row insertions are logged, which can generate many log records in a large bulk copy operation. These log records can be used to both roll forward and roll back the logged bulk copy operation. In a nonlogged bulk copy, only the allocations of new pages to hold the bulk copied rows are logged. This significantly reduces the amount of logging that is needed and speeds the bulk copy operation. If a nonlogged bulk copy operation encounters an error and has to be rolled back, the allocation log records are used to deallocate the pages holding the bulk copied rows. Since the individual row insertions are not logged in a nonlogged bulk copy, however, there is no log record of the individual rows that could be used to roll forward nonlogged bulk copy operations. This is why a nonlogged bulk copy operation invalidates a log backup sequence.

If the database option **trunc. log on chkpt.** is set on, then there is no need to generate log records that would support rolling forward a bulk copy operation. Use nonlogged bulk copy operations in databases where **trunc. log on chkpt.** is turned on.

Whether a bulk copy is logged or nonlogged is not specified as part of the bulk copy operation; it is dependent on the state of the database and the table involved in the bulk copy. A nonlogged bulk copy occurs if all the following conditions are met:

- The database option **select into/bulkcopy** is set to true.
- The target table has no indexes, or if the table has indexes, it is empty when the bulk copy starts.
- The target table is not being replicated.

• The TABLOCK hint is specified using **bcp_control** with *eOption* set to BCPHINTS.

Any bulk copy into SQL Server that does not meet these conditions is logged.

See Also

Logged and Minimally Logged Bulk Copy Operations

Using Data Files and Format Files

The simplest bulk copy program does the following:

- 1. Calls **bcp_init** to specify bulk copying out (set BCP_OUT) from a table or view to a data file.
- 2. Calls **bcp_exec** to execute the bulk copy operation.

The data file is created in native mode; therefore, data from all columns in the table or view are stored in the data file in the same format as in the database. The file can then be bulk copied into a server by using these same steps and setting DB_IN instead of DB_OUT. This works only if both the source and target tables have exactly the same structure. The resulting data file can also be input to the **bcp** utility by using the /**n** (native mode) switch.

To bulk copy out the result set of a Transact-SQL statement instead of directly from a table or view:

- 1. Call **bcp_init** to specify bulk copying out, but specify NULL for the table name.
- 2. Call **bcp_control** with *eOption* set to BCPHINTS and *iValue* set to a pointer to a SQLTCHAR string containing the Transact-SQL statement.
- 3. Call **bcp_exec** to execute the bulk copy operation.

The Transact-SQL statement can be any statement that generates a result set. The data file is created containing the first result set of the Transact-SQL statement. Bulk copy ignores any result set after the first if the Transact-SQL statement generates multiple result sets (for example, if it contains COMPUTE or COMPUTE BY).

To create a data file in which column data is stored in a different format than in the table, call **bcp_columns** to specify how many columns will be changed, then

call **bcp_colfmt** for each column whose format you want to change. This is done after calling **bcp_init** but before calling **bcp_exec**. **bcp_colfmt** specifies the format in which the column's data is stored in the data file. It can be used when bulk copying in or out. You can also use **bcp_colfmt** to set the row and column terminators. For example, if your data contains no tab characters, you can create a tab-delimited file by using **bcp_colfmt** to set the tab character as the terminator for each column.

When bulk copying out and using **bcp_colfmt**, you can easily create a format file describing the data file you have created by calling **bcp_writefmt** after the last call to **bcp_colfmt**.

When bulk copying in from a data file described by a format file, read the format file by calling **bcp_readfmt** after **bcp_init** but before **bcp_exec**.

The **bcp_control** function controls several options when bulk copying into Microsoft® SQL Server[™] from a data file. **bcp_control** sets options, such as the maximum number of errors before termination, the row in the file on which to start the bulk copy, the row to stop on, and the batch size.

To bulk copy by using a format file

Bulk Copying from Program Variables

You can bulk copy directly from program variables. After allocating variables to hold the data for a row and calling **bcp_init** to start the bulk copy, call **bcp_bind** for each column to specify the location and format of the program variable to be associated with the column. Fill each variable with data, then call **bcp_sendrow** to send one row of data to the server. Repeat the process of filling the variables and calling **bcp_sendrow** until all the rows have been sent to the server, then call **bcp_done** to specify that the operation is complete.

The **bcp_bind** *pData* parameter contains the address of the variable being bound to the column. The data for each column can be stored in one of two ways:

- Allocate one variable to hold the data.
- Allocate an indicator variable followed immediately by the data variable.

The indicator variable indicates the length of the data for variable-length columns, and also indicates NULL values if the column allows NULLs. If only a data variable is used, then the address of this variable is stored in the **bcp_bind** *pData* parameter. If an indicator variable is used, the address of the indicator variable is stored in the **bcp_bind** *pData* parameter. The bulk copy functions calculate the location of the data variable by adding the **bcp_bind** *cbIndicator* and *pData* parameters.

bcp_bind supports three methods for dealing with variable-length data:

- Use *cbData* with only a data variable. Place the length of the data in *cbData*. Each time the length of the data to be bulk copied changes, call **bcp_collen** to reset *cbData*. If one of the other two methods is being used, specify SQL_VARLEN_DATA for *cbData*. If all the data values being supplied for a column are NULL, specify SQL_NULL_DATA for *cbData*.
- Use indicator variables. As each new data value is moved into the data variable, store the length of the value in the indicator variable. If one of

the other two methods is being used, specify 0 for *cbIndicator*.

• Use terminator pointers. Load the **bcp_bind** *pTerm* parameter with the address of the bit pattern that terminates the data. If one of the other two methods is being used, specify NULL for *pTerm*.

All three of these methods can be used on the same **bcp_bind** call, in which case the specification that results in the smallest amount of data being copied is used.

The **bcp_bind** *type* parameter uses DB-Library data type identifiers, not ODBC data type identifiers. DB-Library data type identifiers are #defined in Odbcss.h for use with the ODBC **bcp_bind** function.

Bulk copy functions do not support all ODBC C data types. For example, the bulk copy functions do not support the ODBC SQL_C_TYPE_TIMESTAMP structure, so use **SQLBindCol** or **SQLGetData** to convert ODBC SQL_TYPE_TIMESTAMP data to a SQL_C_CHAR variable. If you then use **bcp_bind** with a *type* parameter of SQLCHARACTER to bind the variable to a Microsoft® SQL Server[™] **datetime** column, the bulk copy functions convert the timestamp escape clause in the character variable to the proper datetime format.

Here are the recommended data types to use in mapping from an ODBC SQL data type to a SQL Server data type.

		bcp_bind <i>type</i>	SQL Serve
ODBC SQLdata type	ODBC C data type	parameter	data type
SQL_CHAR	SQL_C_CHAR	SQLCHARACTER	character
			char
SQL_VARCHAR	SQL_C_CHAR	SQLCHARACTER	varchar
			character varying
			char varyiı
			sysname
SQL_LONGVARCHAR	SQL_C_CHAR	SQLCHARACTER	text
SQL_WCHAR	SQL_C_WCHAR	SQLNCHAR	nchar

SQL_WVARCHAR	SQL_C_WCHAR	SQLNVARCHAR	nvarchar
SQL_WLONGVARCHAR	SQL_C_WCHAR	SQLNTEXT	ntext
SQL_DECIMAL	SQL_C_CHAR	SQLCHARACTER	decimal
			dec
			money
			smallmone
SQL_NUMERIC	SQL_C_NUMERIC	SQLNUMERICN	numeric
SQL_BIT	SQL_C_BIT	SQLBIT	bit
SQL_TINYINT (signed)	SQL_C_SSHORT	SQLINT2	smallint
SQL_TINYINT (unsigned)	SQL_C_UTINYINT	SQLINT1	tinyint
SQL_SMALL_INT	SQL_C_SSHORT	SQLINT2	smallint
(signed)			
SQL_SMALL_INT	SQL_C_SLONG	SQLINT4	int
(unsigned)			integer
			litteger
SQL_INTEGER (signed)	SQL_C_SLONG	SQLINT4	int
			integer
SQL_INTEGER	SQL_C_CHAR	SQLCHARACTER	decimal
(unsigned)			dec
SOL DICINT (signed and			higint
SQL_BIGINT (signed and unsigned)	SQL_C_CHAR	SQLCHARACTER	Digilit
SQL_REAL	SQL_C_FLOAT	SQLFLT4	real
SQL_FLOAT	SQL_C_DOUBLE	SQLFLT8	float
SQL_DOUBLE	SQL_C_DOUBLE	SQLFLT8	float
SQL_BINARY	SQL_C_BINARY	SQLBINARY	
SQL_DINAKI	SQL_C_DINARI	SQLDINARI	binary
			timestamp
SQL_VARBINARY	SQL_C_BINARY	SQLBINARY	varbinary
			binary var
SQL_LONGVARBINARY	SQL_C_BINARY	SQLBINARY	image

SQL_TYPE_DATE	SQL_C_CHAR	SQLCHARACTER	datetime smalldateti
SQL_TYPE_TIME	SQL_C_CHAR	SQLCHARACTER	
			smalldatet i
SQL_TYPE_TIMESTAMP	SQL_C_CHAR	SQLCHARACTER	
			smalldatet
SQL_GUID	SQL_C_GUID	SQLUNIQUEID	uniqueider
SQL_INTERVAL_	SQL_C_CHAR	SQLCHARACTER	char

SQL Server does not have signed **tinyint**, unsigned **smallint**, or unsigned **int** data types. To prevent the loss of data values when migrating these data types, create the SQL Server table with the next largest integer data type. To prevent users from later adding values outside the range allowed by the original data type, apply a rule to the SQL Server column to restrict the allowable values to the range supported by the data type in the original source:

```
CREATE TABLE Sample_Ints(STinyIntCol SMALLINT,
USmallIntCol INT)
GO
CREATE RULE STinyInt_Rule
AS
@range >= -128 AND @range <= 127
GO
CREATE RULE USmallInt_Rule
AS
@range >= 0 AND @range <= 65535
GO
sp_bindrule STinyInt_Rule, 'Sample_Ints.STinyIntCol'
GO
sp_bindrule USmallInt_Rule, 'Sample_Ints.USmallIntCol'
GO
```

SQL Server does not support interval data types directly. An application can, however, store interval escape sequences as character strings in a SQL Server character column. The application can read them for later use, but they cannot be used in Transact-SQL statements.

The bulk copy functions can be used to quickly load data into SQL Server that has been read from an ODBC data source. Use **SQLBindCol** to bind the columns of a result set to program variables, then use **bcp_bind** to bind the same program variables to a bulk copy operation. Calling **SQLFetchScroll** or **SQLFetch** then fetches a row of data from the ODBC data source into the program variables, and calling **bcp_sendrow** bulk copies the data from the program variables to SQL Server.

An application can use the **bcp_colptr** function anytime it needs to change the address of the data variable originally specified in the **bcp_bind** *pData* parameter. An application can use the **bcp_collen** function anytime it needs to change the data length originally specified in the **bcp_bind** *cbData* parameter.

You cannot read data from SQL Server into program variables using bulk copy; there is nothing like a "bcp_readrow" function. You can only send data from the application to the server.

To bulk copy data from program variables

Managing Bulk Copy Batch Sizes

The primary purpose of a batch in bulk copy operations is to define the scope of a transaction. If a batch size is not set, then bulk copy functions consider an entire bulk copy to be one transaction. If a batch size is set, then each batch constitutes a transaction that is committed when the batch finishes.

If a bulk copy is performed with no batch size specified and an error is encountered, the entire bulk copy is rolled back. The recovery of a long-running bulk copy can take a long time. When a batch size is set, bulk copy considers each batch a transaction and commits each batch. If an error is encountered, only the last outstanding batch needs to be rolled back.

The batch size can also affect locking overhead. When performing a bulk copy against Microsoft® SQL Server[™], the TABLOCK hint can be specified using **bcp_control** to acquire a table lock instead of row locks. The single table lock can be held with minimal overhead for an entire bulk copy operation. If TABLOCK is not specified then locks are held on individual rows and the overhead of maintaining all the locks for the duration of the bulk copy can slow performance. Because locks are only held for the length of a transaction, specifying a batch size addresses this problem by periodically generating a commit that frees the locks currently held.

The number of rows making up a batch can have significant performance effects when bulk copying a large number of rows. The recommendations for batch size depend on the type of bulk copy being performed.

- When bulk copying to SQL Server, specify the TABLOCK bulk copy hint and set a large batch size.
- When TABLOCK is not specified, limit batch sizes to less than 1,000 rows.

When bulk copying in from a data file, the batch size is specified by calling **bcp_control** with the BCPBATCH option before calling **bcp_exec**. When bulk copying from program variables using **bcp_bind** and **bcp_sendrow**, the batch size is controlled by calling **bcp_batch** after calling **bcp_sendrow** *x* times,

where *x* is the number of rows in a batch.

In addition to specifying the size of a transaction, batches also affect when rows are sent across the network to the server. Bulk copy functions normally cache the rows from **bcp_sendrow** until a network packet is filled, and then send the full packet to the server. When an application calls **bcp_batch**, however, the current packet is sent to the server regardless of whether it has been filled. Using a very low batch size can slow performance if it results in sending many partially filled packets to the server. For example, calling **bcp_batch** after every **bcp_sendrow** causes each row to be sent in a separate packet and, unless the rows are very large, wastes space in each packet. The default size of network packets for SQL Server is 4 KB, although an application can change the size by calling **SQLSetConnectAttr** specifying the SQL_ATTR_PACKET_SIZE attribute.

Another side effect of batches is that each batch is considered an outstanding result set until it is completed with **bcp_batch**. If any other operations are attempted on a connection handle while a batch is outstanding, the SQL Server ODBC driver issues an error with SQLState = "HY000" and an error message string of:

"[Microsoft][ODBC SQL Server Driver] Connection is busy with results for another hstmt."

See Also

bcp_batch bcp_control SQLSetConnectAttr Batch Switches

Bulk Copying text and image Data

Large **text**, **ntext**, and **image** values are bulk copied using the **bcp_moretext** function. You code **bcp_bind** for the **text**, **ntext**, or **image** column with a *pData* pointer set to NULL indicating the data will be provided with **bcp_moretext**. It is important to specify the exact length of data supplied for each **text**, **ntext**, or **image** column in each bulk-copied row. If the length of the data for a column is different from the column length specified in **bcp_bind**, use **bcp_collen** to set the length to the proper value. A **bcp_sendrow** sends all the non-**text**, non-**ntext**, and non-**image** data; you then call **bcp_moretext** to send the **text**, **ntext**, or **image** data in separate units. Bulk copy functions determine that all data has been sent for the current **text**, **ntext**, or **image** column when the sum of the lengths of data sent through **bcp_moretext** equals the length specified in the latest **bcp_collen** or **bcp_bind**.

bcp_moretext has no parameter to identify a column. When there are multiple
text, ntext, or image columns in a row, bcp_moretext operates on the text,
ntext, or image columns starting with the column having the lowest ordinal
number and proceeding to the column with the highest ordinal number.
bcp_moretext goes from one column to the next when the sum of the lengths of
data sent equals the length specified in the latest bcp_collen or bcp_bind for the
current column.

See Also

bcp_bind bcp_collen bcp_moretext bcp_sendrow

Converting from DB-Library to ODBC Bulk Copy

Converting a DB-Library bulk copy program to ODBC is easy because the bulk copy functions supported by the Microsoft® SQL Server[™] ODBC driver are similar to the DB-Library bulk copy functions, with the following exceptions:

- DB-Library applications pass a pointer to a DBPROCESS structure as the first parameter of bulk copy functions. In ODBC applications, the DBPROCESS pointer is replaced with an ODBC connection handle.
- DB-Library applications call **BCP_SETL** before connecting to enable bulk copy operations on a DBPROCESS. ODBC applications instead call **SQLSetConnectAttr** before connecting to enable bulk operations on a connection handle:

SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (void *)SQL_BCP_ON, SQL_IS_INTEGER);

- The SQL Server ODBC driver does not support DB-Library message and error handlers; you must call **SQLGetDiagRec** to get errors and messages raised by the ODBC bulk copy functions. The ODBC versions of bulk copy functions return the standard bulk copy return codes of SUCCEED or FAILED, not ODBC-style return codes, such as SQL_SUCCESS or SQL_ERROR.
- The values specified for the DB-Library **bcp_bind** *varlen* parameter are interpreted differently than the ODBC **bcp_bind** *cbData* parameter.

Condition	DB-Library varlen	
indicated	value	ODBC <i>cbData</i> value
Null values	0	-1 (SQL_NULL_DATA)
supplied		
Variable data	-1	-10
supplied		(SQL_VARLEN_DATA)
Zero length	NA	0

character or binary	
string	

In DB-Library, a *varlen* value of -1 indicates that variable length data is being supplied, which in the ODBC *cbData* is interpreted to mean that only NULL values are being supplied. Change any DB-Library *varlen* specifications of -1 to SQL_VARLEN_DATA and any *varlen* specifications of 0 to SQL_NULL_DATA.

- The DB-Library **bcp_colfmt** *file_collen* and the ODBC **bcp_colfmt** *cbUserData* have the same issue as the **bcp_bind** *varlen* and *cbData* parameters noted above. Change any DB-Library *file_collen* specifications of -1 to SQL_VARLEN_DATA and any *file_collen* specifications of 0 to SQL_NULL_DATA.
- The *iValue* parameter of the ODBC **bcp_control** function is a void pointer. In DB-Library, *iValue* was an integer. Cast the values for the ODBC *iValue* to void *.
- The **bcp_control** option BCPMAXERRS specifies how many individual rows can have errors before a bulk copy operation fails. The default for BCPMAXERRS is 0 (fail on first error) in the DB-Library version of **bcp_control** and 10 in the ODBC version. DB-Library applications that depend on the default of 0 to terminate a bulk copy operation must be changed to call the ODBC **bcp_control** to set BCPMAXERRS to 0.
- The ODBC **bcp_control** function supports the following options not supported by the DB-Library version of **bcp_control**:
 - BCPODBC

When set to TRUE, specifies that **datetime** and **smalldatetime** values saved in character format will have the ODBC timestamp escape sequence prefix and suffix. This only applies to BCP_OUT operations.

With BCPODBC set to FALSE, a **datetime** value converted to a character string is output as:

1997-01-01 00:00:00.000

With BCPODBC set to TRUE, the same **datetime** value is output as:

{ts '1997-01-01 00:00:00.000' }

• BCP6xFILEFMT

When set to TRUE, specifies that program variables bound to columns in native format, or columns stored in an operating-system file in either native or character format use the SQL Server version 6*x* format instead of the SQL Server 7.0 format. Columns bound in native format include columns for which **bcp_bind** was called with *type* set to 0 and *varlen* set to SQL_VARLEN_DATA. Columns stored in native format in a data file include columns for which **bcp_colfmt** was called with *file_type* set to 0 and *file_collen* set to SQL_VARLEN_DATA.

• BCPKEEPIDENTITY

When set to TRUE, specifies that bulk copy functions insert data values supplied for columns with identity constraints. If this is not set, new identity values are generated for the inserted rows.

• BCPHINTS

Specifies various bulk copy optimizations. This option cannot be used on 6.5 or earlier versions of SQL Server.

• BCPFILECP

Specifies the code page of the bulk copy file.

• BCPUNICODEFILE

Specifies that a character mode bulk copy file is a Unicode file.

- The ODBC **bcp_colfmt** function does not support the *file_type* indicator of SQLCHAR because it conflicts with the ODBC SQLCHAR typedef. Use SQLCHARACTER instead for **bcp_colfmt**.
- In the ODBC versions of bulk copy functions, the format for working with **datetime** and **smalldatetime** values in character strings is the ODBC format of yyyy-mm-dd hh:mm:ss.sss; **smalldatetime** values use the ODBC format of yyyy-mm-dd hh:mm:ss.

The DB-Library versions of the bulk copy functions accept **datetime** and **smalldatetime** values in character strings using several formats:

- The default format is *mmm dd yyyy hh:mmxx* where *xx* is either AM or PM.
- **datetime** and **smalldatetime** character strings in any format supported by the DB-Library **dbconvert** function.
- When the **Use international settings** box is checked on the DB-Library **Options** tab of the SQL Server Client Network Utility, the DB-Library bulk copy functions also accept dates in the regional date format defined for the locale setting of the client computer registry.

The DB-Library bulk copy functions do not accept the ODBC **datetime** and **smalldatetime** formats.

The ODBC bulk copy functions will accept the same **datetime** formats as the DB-Library versions of the bulk copy functions when BCP6xFILEFMT is specified. If the SQL_SOPT_SS_REGIONALIZE statement attribute is set to SQL_RE_ON, the ODBC bulk copy functions will accept dates in the regional date format defined for the locale setting of the client computer registry.

• When outputting **money** values in character format, ODBC bulk copy functions supply 4 digits of precision and no comma separators; DB-Library versions only supply 2 digits of precision and include the comma separators.

See Also

bcp_colfmtbcp_controlBulk-Copy FunctionsSQLSetConnectAttrUsing Format Files

Managing text and image Columns

Microsoft® SQL Server[™] **text**, **ntext**, and **image** data (also referred to as long data) are character or binary string data types that can hold data values too large to fit into **char**, **varchar**, **binary**, or **varbinary** columns. The SQL Server **text** data type maps to the ODBC SQL_LONGVARCHAR data type; **ntext** maps to SQL_WLONGVARCHAR; and **image** maps to SQL_LONGVARBINARY. Some data items, such as long documents or large bitmaps, may be too large to store reasonably in memory. To retrieve long data from SQL Server in sequential parts, the SQL Server ODBC driver enables an application to call **SQLGetData**. To send long data in sequential parts, the application can call **SQLPutData**. Parameters for which data is sent at execution time are known as data-at-execution parameters.

An application can actually write or retrieve any type of data (not just long data) with **SQLPutData** or **SQLGetData**, although only **character** and **binary** data can be sent or retrieved in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use **SQLPutData** or **SQLGetData**. It is much easier to bind the single buffer to the parameter or column.

To use data-at-execution parameters

Bound vs. Unbound text and image Columns

When using server cursors, the Microsoft® SQL ServerTM ODBC driver is optimized to not transmit the data for unbound **text**, **ntext**, or **image** columns at the time **SQLFetch** is performed. The **text**, **ntext**, or **image** data is not actually retrieved from the server until the application issues **SQLGetData** for the column.

Many applications can be written so that no **text**, **ntext**, or **image** data is displayed while a user is simply scrolling up and down in a cursor. When a user selects a row to get more detail, the application can then call **SQLGetData** to retrieve the **text**, **ntext**, or **image** data. This will prevent transmitting the **text**, **ntext**, or **image** data for any of the rows the user does not select, and can therefore prevent the transmission of very large amounts of data.

To use data-at-execution parameters

Logged vs. Unlogged Modifications

An application can request that the Microsoft® SQL Server[™] ODBC driver not log **text**, **ntext**, and **image** modifications. Care should be used with this option, however. It should be used only for those situations where the **text**, **ntext**, or **image** data is not critical and data owners are willing to trade off the ability to recover data for higher performance.

The logging of **text**, **ntext**, and **image** modifications is controlled by calling **SQLSetStmtAttr** with the *Attribute* parameter set to SQL_SOPT_SS_ TEXTPTR_LOGGING and *ValuePtr* set to either SQL_TL_ON or SQL_TL_OFF.

See Also

<u>SQLSetStmtAttr</u>

Data-at-execution and text, ntext, or image Columns

ODBC data-at-execution is a feature that enables applications to work with extremely large amounts of data on bound columns or parameters. When retrieving very large **text**, **ntext**, or **image** columns, an application may not be able to simply allocate a huge buffer, bind the column into the buffer, and fetch the row. When updating very large **text**, **ntext**, or **image** columns, the application may not be able to simply allocate a huge buffer, bind it to a parameter marker in an SQL statement, and then execute the statement. In these cases, the application must use **SQLGetData** or **SQLPutData** with its data-atexecution options.

To use data-at-execution parameters

Connecting to a Failover Server

The Microsoft® SQL Server[™] ODBC driver supports a failover configuration using the driver-specific SQL_FALLBACK_CONNECT connection option for **SQLSetConnectAttr** and **SQLGetConnectAttr**. If the server you are connecting to has a failover server, the driver can connect to the failover server if the primary server is unavailable.

ODBC applications can take advantage of SQL Server's failover feature by calling **SQLSetConnectAttr** with SQL_FALLBACK_CONNECT enabled before connecting. When the driver connects to the primary server, it retrieves all the information it needs to connect to the failover server and stores the information in the client registry. If the application then loses its connect to the primary server, it completes its current transaction and attempts to reconnect to the primary server. If unsuccessful, it uses the registry information to attempt to connect to the failover server.

See Also

SQLGetConnectAttr SQLSetConnectAttr

Profiling ODBC Driver Performance

The Microsoft® SQL Server[™] ODBC driver can profile two types of performance data:

• Long-running queries.

The driver can write to a log file any query that does not get a response from the server within a specified amount of time. Application programmers or database administrators can then research each logged SQL statement to determine how they can improve its performance.

• Driver-performance data.

The driver can record performance statistics and either write them to a file or make them available to an application through a driver-specific data structure named SQLPERF. The file containing the performance statistics is a tab-delimited file that can be easily analyzed with any spreadsheet that supports tab-delimited files, such as Microsoft Excel.

Either type of profiling can be turned on by:

- Connecting to a data source that specifies logging.
- Calling **SQLSetConnectAttr** to set driver-specific attributes that control profiling.

Each application process gets its own copy of the SQL Server ODBC driver, and profiling is global to the combination of a driver copy and an application process. When anything in the application turns on profiling, profiling records information for all connections active in the driver from that application. Even connections that did not specifically call for profiling are included.

After the driver has opened a profiling log (either the performance data or longrunning query log), it does not close the log until the driver is unloaded by the ODBC Driver Manager, when an application frees all the environment handles it opened in the driver. If the application opens a new environment handle, a new copy of the driver is loaded. If the application then either connects to a data source that specifies the same log file or sets the driver-specific attributes to log to the same file, the driver overwrites the old log.

If an application starts profiling to a log file and a second application attempts to start profiling to the same log file, the second application is not able to log any profiling data. If the second application starts profiling after the first application has unloaded its driver, the second application overwrites the log file from the first application.

If an application connects to a data source that has profiling enabled, the driver returns SQL_ERROR if the application calls **SQLSetConnectOption** to start logging. A call to **SQLGetDiagRec** then returns:

SQLState: 01000, pfNative = 0 ErrorMsg: [Microsoft][ODBC SQL Server Driver] An error has occurred during the attempt to access the log file, logging disabled.

The driver stops gathering performance data when an environment handle is closed. If an ODBC 3.*x* application has multiple connections, each with its own environment handle, then the driver will stop gathering performance data when any of the associated environment handles are closed.

The driver's performance data can either be stored in the SQLPERF data structure or logged in a tab-delimited file. The data includes the following categories of statistics:

- Application profile
- Connection
- Network
- Time

In the following table, the descriptions of the fields in the SQLPERF data structure also apply to the statistics recorded in the performance log file.

SQLPERF Field	Description		
Application Profile Statistics:			
TimerResolution	Minimum resolution of the server's clock time in milliseconds. This is usually reported as 0 (zero) and should only be considered if the number reported is large. If the minimum resolution of the server clock is larger than the likely interval for some of the timer-based statistics, those statistics could be inflated.		
SQLidu	Number of INSERT, DELETE, or UPDATE statements after SQL_PERF_START.		
SQLiduRows	Number of INSERT, DELETE, or UPDATE statements after SQL_PERF_START.		
SQLSelects	Number of SELECT statements processed after SQL_PERF_START.		
SQLSelectRows	Number of rows selected after SQL_PERF_START.		
Transactions	Number of user transactions after SQL_PERF_START, including rollbacks. When an ODBC application is running with SQL_AUTOCOMMIT_ON, each command is considered a transaction.		
SQLPrepares	Number of SQLPrepare calls after SQL_PERF_START.		
ExecDirects	Number of SQLExecDirect calls after SQL_PERF_START.		
SQLExecutes	Number of SQLExecute calls after SQL_PERF_START.		
CursorOpens	Number of times the driver has opened a server cursor after SQL_PERF_START.		
CursorSize	Number of rows in the result sets opened by cursors after SQL_PERF_START.		
CursorUsed	Number of rows actually retrieved through the driver from cursors after SQL_PERF_START.		
PercentCursorUsed	Equals CursorUsed/CursorSize. For example, if		

	an application causes the driver to open a server	
	cursor to do "SELECT COUNT(*) FROM authors," 23 rows will be in the result set for the	
	SELECT statement. If the application then	
	fetches only three of these rows,	
	CursorUsed/CursorSize is 3/23, so	
	PercentCursorUsed is 13.043478.	
AvgFetchTime	Equals SQLFetchTime/SQLFetchCount.	
AvgCursorSize	Equals CursorSize/CursorOpens.	
AvgCursorUsed	Equals CursorUsed/CursorOpens.	
SQLFetchTime	Cumulative amount of time it took fetches against server cursors to complete.	
SQLFetchCount	Number of fetches done against server cursors after SQL_PERF_START.	
CurrentStmtCount	Number of statement handles currently open on	
	all connections open in the driver.	
MaxOpenStmt	Maximum number of concurrently opened statement handles after SQL_PERF_START.	
SumOpenStmt	Number of statement handles that have been	
-	opened after SQL_PERF_START.	
Connection Statistics:		
CurrentConnectionCount	Current number of active connection handles the	
	application has open to the server.	
MaxConnectionsOpened	Maximum number of concurrent connection handles opened after SQL_PERF_START.	
SumConnectionsOpened	Sum of the number of connection handles that have been opened after SQL_PERF_START.	
SumConnectionTime	Sum of the amount of time that all of the	
	connections have been opened after	
	SQL_PERF_START. For example, if an	
	application opened 10 connections and	
	maintained each connection for 5 seconds, then	
	SumConnectionTime would be 50 seconds.	
AvgTimeOpened	Equals SumConnectionsOpened/	
	SumConnectionTime.	

Network Statistics:		
ServerRndTrips	The number of times the driver sent commands to the server and got a reply back.	
BuffersSent	Number of Tabular Data Stream (TDS) packets sent to SQL Server by the driver after SQL_PERF_START. Large commands can take multiple buffers, so if a large command is sent to the server and it fills six packets, ServerRndTrips is incremented by one and BuffersSent is incremented by six.	
BuffersRec	Number of TDS packets received by the driver from SQL Server after the application started using the driver.	
BytesSent	Number of bytes of data sent to SQL Server in TDS packets after the application started using the driver.	
BytesRec Time Statistics:	Number of bytes of data in TDS packets received by the driver from SQL Server after the application started using the driver.	
msExecutionTime	Cumulative amount of time the driver spent processing after SQL_PERF_START, including the time spent waiting for replies from the server.	
msNetworkServerTime	Cumulative amount of time the driver spent waiting for replies from the server.	

To profile driver performance data

SQL Server ODBC Driver Programmer's Reference

Open Database Connectivity (ODBC) is a Microsoft® Win32® API used by applications to access data in ODBC data sources.

The SQL Server ODBC Driver Programmer's Reference does not document all of the ODBC function calls. Only those functions that have driver-specific parameters or behaviors when used with the Microsoft® SQL Server[™] ODBC driver are discussed. The functions documented in the SQL Server ODBC Driver Programmer's Reference use ODBC 3.5.

For a full description of the ODBC API, see the Microsoft ODBC Software Development Kit (SDK). The ODBC SDK is part of the Microsoft Developer Network (MSDN®). The ODBC SDK can also be downloaded from the <u>Microsoft Web site</u>, and is available in the *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference* available from Microsoft Press®.

ODBC API Implementation Details

This section documents the ODBC functions that exhibit SQL Server-specific behaviors when used with the Microsoft® SQL Server[™] ODBC driver. Not all ODBC functions are documented here. The individual topics only discuss the SQL Server-specific issues for an ODBC function. They are not a complete reference for the ODBC function.

The SQL Server 2000 ODBC driver complies with the ODBC 3.51 specification. For a comprehensive reference of ODBC 3.51, download the Microsoft Data Access SDK from the <u>Microsoft Web site</u>. The *ODBC 3.0 Software Development Kit and Programmer's Reference* is also available from Microsoft Press®.

SQLBindCol

As a general rule, consider the implications of using **SQLBindCol** to cause data conversion. Binding conversions are client processes, so, for example, retrieving a floating-point value bound to a character column causes the driver to perform the float-to-character conversion locally when a row is fetched. The Transact-SQL CONVERT function can be used to place the cost of data conversion on the server.

An instance of Microsoft® SQL Server[™] can return multiple sets of result rows on a single statement execution. Each result set must be bound separately. For more information about binding for multiple result sets, see <u>SQLMoreResults</u>.

The developer can bind columns to SQL Server-specific C data types using the *TargetType* value SQL_C_BINARY. Columns bound to SQL Server-specific types are not portable. The defined SQL Server-specific ODBC C data types match the type definitions for DB-Library, and DB-Library developers porting applications may want to take advantage of this feature.

Reporting data truncation is an expensive process for the SQL Server ODBC driver. You can avoid truncation by ensuring that all bound data buffers are wide enough to return data. For character data, the width should include space for a string terminator when the default driver behavior for string termination is used. For example, binding an SQL Server **char(5)** column to an array of five characters results in truncation for every value fetched. Binding the same column to an array of six characters avoids the truncation by providing a character element in which to store the null terminator. **SQLGetData** can be used to efficiently retrieve long character and binary data without truncation.

See Also

<u>SQLGetData</u>

SQLBindParameter

SQLBindParameter can eliminate the burden of data conversion when used to provide data for the Microsoft® SQL Server[™] ODBC driver, resulting in significant performance gains for both the client and server components of applications. Other benefits include reduced loss of precision when inserting or updating approximate numeric data types.

If the SQL Server ODBC driver encounters an error on a single array element of an array of parameters, the driver continues to execute the statement for the remaining array elements. If the application has bound an array of parameter status elements for the statement, the row(s) of parameters generating errors can be determined from the array.

When using the SQL Server ODBC driver version 3.7 or later, specify SQL_PARAM_INPUT when binding input parameters. Only specify SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT when binding stored procedure parameters defined with the OUTPUT keyword.

SQLRowCount is unreliable with the SQL Server ODBC driver if an array element of a bound-parameter array causes an error in statement execution. The ODBC statement attribute SQL_ATTR_PARAMS_PROCESSED_PTR will report the number of rows processed prior to the error occurring. The application can then traverse its parameter status array to discover the number of statements successfully executed, if necessary.

SQLBrowseConnect

SQLBrowseConnect uses keywords that can be categorized into three levels of connection information. For each keyword, the following table indicates whether a list of valid values is returned and whether the keyword is optional.

Level 1

	List		
Keyword	returned?	Optional?	Description
DSN	N/A	No	Name of the data source returned by
			SQLDataSources. The DSN keyword
			cannot be used if the DRIVER
			keyword is used.
DRIVER	N/A	No	Microsoft [®] SQL Server [™] ODBC
			driver name is {SQL Server} or SQL
			Server (braces are required when using
			driver version 2.65 or earlier). The
			DRIVER keyword cannot be used if
			the DSN keyword is used.

Level 2

	List		
Keyword	returned?	Optional?	Description
SERVER	Yes		Name of the server on the network on which the data source resides. When running on Microsoft Windows NT® 4.0, "(local)" can be entered as the server, in which case a local copy of SQL Server can be used, even when this is a nonnetworked version.
UID	No	Yes	User login ID.
PWD	No	Yes (depends on	User-specified password.

		the user)	
APP	No	Yes	Name of the application calling
			SQLBrowseConnect.
WSID	No		Workstation ID. Typically, this is the network name of the computer on which the application runs.

Level 3

	List		
Keyword	returned?	Optional?	Description
DATABASE	Yes	Yes	Name of the SQL Server database.
LANGUAGE	Yes	Yes	National language used by SQL
			Server.

SQLBrowseConnect ignores the values of the DATABASE and LANGUAGE keywords stored in the ODBC data source definitions. If the database or language specified in the connection string passed to **SQLBrowseConnect** is invalid, **SQLBrowseConnect** returns SQL_NEED_DATA and the level 3 connection attributes.

SQLBrowseConnect does not verify user access to all the databases listed with the DATABASE keyword when connected to SQL Server version 6.5 or earlier servers. If the user does not have access to the chosen database, **SQLBrowseConnect** returns SQL_NEED_DATA and the level 3 connection attributes.

The following attributes, set by calling **SQLSetConnectAttr**, determine the result set returned by **SQLBrowseConnect**.

SQL_COPT_SS_BROWSE_CONNECT: If it is set to SQL_MORE_INFO_NO, in SQL Server version 6.5 and later, **SQLBrowseConnect** returns a list of servers. If it is set to SQL_MORE_INFO_YES, in SQL Server version 6.5 and 7.0, **SQLBrowseConnect** returns a list of servers. In SQL Server 2000, **SQLBrowseConnect** returns an extended string of server properties.

This is an example of an extended string returned by **SQLBrowseConnect** in SQL Server 2000:

ServerName\InstanceName;Clustered:No;Version:8.00.131

In this string, semi-colons separate various parts of information about the server, and commas separate different server instances.

SQL_COPT_SS_BROWSE_SERVER: If a server name is specified, **SQLBrowseConnect** will return information for the server specified. If SQL_COPT_SS_BROWSE_SERVER is set to NULL, **SQLBrowseConnect** returns information for all servers in the domain.

SQLCloseCursor

SQLCloseCursor replaces **SQLFreeStmt** with an *Option* value of SQL_CLOSE. On receipt of **SQLCloseCursor**, the Microsoft® SQL ServerTM ODBC driver discards pending result set rows. Note that the statement's column and parameter bindings (if any exist) are left unaltered by **SQLCloseCursor**.

SQLColAttribute

You can use **SQLColAttribute** to retrieve an attribute of a result set column for either prepared or executed ODBC statements. Calling **SQLColAttribute** on prepared statements causes a roundtrip to the Microsoft® SQL Server[™]. The SQL Server ODBC driver receives result set column data as part of statement execution, so calling **SQLColAttribute** after the completion of **SQLExecute** or **SQLExecDirect** does not involve a server roundtrip.

FieldIdentifier value	Description
SQL_COLUMN_TABLE_NAME	Available on result sets retrieved
	from statements that generate
	server cursors or on executed
	SELECT statements containing a
	FOR BROWSE clause.
SQL_DESC_BASE_COLUMN_NAME	Available on result sets retrieved
	from statements that generate
	server cursors or on executed
	SELECT statements containing a
	FOR BROWSE clause.
SQL_DESC_BASE_TABLE_NAME	Available on result sets retrieved
	from statements that generate
	server cursors or on executed
	SELECT statements containing a
	FOR BROWSE clause.
SQL_DESC_CATALOG_NAME	Database name. Available on
	result sets retrieved from
	statements that generate server
	cursors or on executed SELECT
	statements containing a FOR
	BROWSE clause.
SQL_DESC_LABEL	Available on all result sets. The
	value is identical to the value of

ODBC column identifier attributes are not available on all SQL Server result sets.

	the SQL_DESC_NAME field.
	The field is zero length only if a column is the result of an expression and the expression does not contain a label assignment.
SQL_DESC_NAME	Available on all result sets. The value is identical to the value of the SQL_DESC_LABEL field.
	The field is zero length only if a column is the result of an expression and the expression does not contain a label assignment.
SQL_DESC_SCHEMA_NAME	Owner name. Available on result sets retrieved from statements that generate server cursors or on executed SELECT statements containing a FOR BROWSE clause.
	Available only if the owner name is specified for the column in the SELECT statement.
SQL_DESC_TABLE_NAME	Available on result sets retrieved from statements that generate server cursors or on executed SELECT statements containing a FOR BROWSE clause.
SQL_DESC_UNNAMED	SQL_NAMED for all columns in a result set unless a column is the result of an expression that does not contain a label assignment as part of the expression. When

SQL_DESC_UNNAMED returns
SQL_UNNAMED, all ODBC
column identifier attributes
contain zero length strings for the
column.

Note When connected to an instance of SQL Server 4.2*x*, **SQLColAttribute** must create a result set to report column attributes. The SQL Server ODBC driver appends the clause WHERE 1 = 2 to prepared SELECT statements prior to execution. When connected to SQL Server 4.2*x*, **SQLColAttribute** cannot return information about a result set that is generated by a procedure if that procedure has been prepared but not executed.

When connected to any later version of SQL Server, the SQL Server ODBC driver uses the SET FMTONLY statement to reduce server overhead when **SQLColAttribute** is called for prepared but unexecuted statements.

For all versions, column attributes are reported for only the first result set when multiple result sets are generated by a prepared batch of SQL statements.

The following column attributes are extensions exposed by the SQL Server ODBC driver. The SQL Server ODBC driver returns all values in the *NumericAttrPtr* parameter. The values are returned as SDWORD (signed long) except SQL_CA_SS_COMPUTE_BYLIST, which is a pointer to a WORD array.

Value returned
TRUE if the column referenced is
part of a hidden primary key created
to support a Transact-SQL SELECT
statement containing FOR
BROWSE.
Ordinal position of a COMPUTE
clause result column within the
current Transact-SQL SELECT
statement.
TRUE if the column referenced is
part of a primary key for the row and
the Transact-SQL SELECT

	statement contains FOR BROWSE.
SQL_CA_SS_COLUMN_OP	Integer specifying the aggregate operator responsible for the value in a COMPUTE clause column. Definitions of the integer values are in Odbcss.h.
SQL_CA_SS_COLUMN_ORDER	Ordinal position of the column within an ODBC or Transact-SQL SELECT statement's ORDER BY clause.
SQL_CA_SS_COLUMN_SIZE	Maximum length, in bytes, required to bind a data value retrieved from the column to a SQL_C_BINARY variable.
SQL_CA_SS_COLUMN_SSTYPE	Native data type of data stored in the SQL Server column. Definitions of the type values are in Odbcss.h.
SQL_CA_SS_COLUMN_UTYPE	Base data type of the SQL Server column's user-defined data type. Definitions of the type values are in Odbcss.h.
SQL_CA_SS_COLUMN_VARYLEN	TRUE if the column's data can vary in length, FALSE otherwise.
SQL_CA_SS_COMPUTE_BYLIST	Pointer to an array of WORD (unsigned short) specifying the columns used in the BY phrase of a COMPUTE clause. If the COMPUTE clause does not specify a BY phrase, a NULL pointer is returned.
	The first element of the array contains the count of BY list columns. Additional elements are the column ordinals.
SQL_CA_SS_COMPUTE_ID	<i>computeid</i> of a row that is the result

	of a COMPUTE clause in the current Transact-SQL SELECT
	statement.
SQL_CA_SS_NUM_COMPUTES	Number of COMPUTE clauses
	specified in the current Transact-
	SQL SELECT statement.
SQL_CA_SS_NUM_ORDERS	Number of columns specified in an
	ODBC or Transact-SQL SELECT
	statement's ORDER BY clause.
* Available if statement attribute SQL_SOPT_SS_HIDDEN_COLUMNS	
is set to SQL_HC_ON.	

See Also

<u>SQLSetStmtAttr</u>

SQLColumnPrivileges

SQLColumnPrivileges uses the catalog stored procedure **sp_column_privileges** to report user permissions for columns in a table.

The following table shows **SQLColumnPrivileges** parameter mapping for **sp_column_privileges** stored procedure execution.

SQLColumnPrivileges parameter	sp_column_privileges parameter
name	name
CatalogName	table_qualifier
SchemaName	table_owner
TableName	table_name
ColumnName	column_name

SQLColumnPrivileges returns SQL_SUCCESS whether or not values exist for the *CatalogName, SchemaName, TableName*, or *ColumnName* parameters. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLColumnPrivileges can be executed on a static server cursor. An attempt to execute **SQLColumnPrivileges** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

The Microsoft® SQL Server[™] ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *CatalogName* parameter: *Linked_Server_Name.Catalog_Name*.

See Also

sp_column_privileges

SQLColumns

SQLColumns executes the Transact-SQL procedure **sp_columns** to report catalog data for database columns.

The following table shows **SQLColumns** parameter mapping for **sp_columns** stored procedure execution.

SQLColumns parameter name	sp_columns parameter name
CatalogName	object_qualifier
SchemaName	object_owner
TableName	object_name
ColumnName	column_name

SQLColumns returns SQL_SUCCESS whether or not values exist for the *CatalogName*, *TableName*, or *ColumnName* parameters. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLColumns can be executed on a static server cursor. An attempt to execute **SQLColumns** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

The Microsoft® SQL Server[™] ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *CatalogName* parameter: *Linked_Server_Name.Catalog_Name*.

For ODBC 2.*x* applications not using wildcards in *TableName*, **SQLColumns** returns information about any tables whose names match *TableName* and are owned by the current user. If the current user owns no table whose name matches the *TableName* parameter, **SQLColumns** returns information about any tables owned by other users where the table name matches the *TableName* parameter. For ODBC 2.*x* applications using wildcards, **SQLColumns** returns all tables whose names match *TableName*. For ODBC 3.*x* applications **SQLColumns** returns all tables whose names match *TableName* regardless of owner or whether wildcards are used.

See Also

<u>sp_columns</u>

SQLConfigDataSource

The Microsoft® SQL Server[™] ODBC driver supports the following SQL Server-specific keyword/value pairs for data source configuration attribute strings.

Keyword	Values	Description
Address		Network address of the SQL Server.
AnsiNPW	yes	Default. Specifies that ANSI_NULLS, ANSI_WARNINGS, ANSI_PADDING, and CONCAT_NULL_YIELDS_NULL are set ON for each connection. This allows SQL Server to treat SQL statements as per SQL-92. For more information see Effects of SQL-92 Options.
	no	Do not use ANSI-defined behaviors for NULL comparisons, padding, warnings, and NULL concatenation.
AttachDBFileName	file_path	Name of the primary file of an attachable database. Include the full path, and escape any \ characters if using a C character string variable: AttachDBFileName=c:\\AB\\MyDB.mdf This database is attached and becomes the default database for the connection. To use AttachDBFileName you must also specify the database name in either the SQLDriverConnect DATABASE parameter or the SQL_COPT_CURRENT_CATALOG connection attribute. If the database was previously attached, SQL Server will not reattach it; it will use the attached

		database as the default for the connection.
AutoTranslate	yes	Default. ANSI character strings sent between the client and server are translated by converting through Unicode to minimize problems in matching extended characters between the code pages on the client and the server:
		Client SQL_C_CHAR data sent to a SQL Server char , varchar , or text variable, parameter, or column is converted from character to Unicode using the client ANSI code page (ACP), then converted from Unicode to character using the ACP of the server.
		SQL Server char , varchar , or text data sent to a client SQL_C_CHAR variable is converted from character to Unicode using the server ACP, then converted from Unicode to character using the client ACP.
		These conversions are performed on the client by the SQL Server ODBC driver. This requires that the same ANSI code page (ACP) used on the server be available on the client.
		These settings have no effect on the conversions that occur for these transfers:
		Unicode SQL_C_WCHAR client data sent to char , varchar , or text on the server.

		char , varchar , or text server data sent to a Unicode SQL_C_WCHAR variable on the client.
		ANSI SQL_C_CHAR client data sent to Unicode nchar , nvarchar , or ntext on the server.
		Unicode char , varchar , or text server data sent to an ANSI SQL_C_CHAR variable on the client.
	no	Do not perform character translation. The SQL Server ODBC driver does not translate client ANSI character data sent to char , varchar , or text variables, parameters, or columns on the server. No translation is performed on char , varchar , or text data sent from the server to character variables on the client. If the client and SQL Server installation
		are using different ACPs, then extended characters can be misinterpreted.
Database		Name of the default database for the connection. If Database is not specified, the default database defined for the login is used. The default database from the ODBC data source overrides the default database defined for the login. If AttachDBFileName points to a primary database file, the database is attached and given the name specified in Database.
Description		Descriptive text. The description appears with the data source in the ODBC Management utility.

Driver	{SQL Server}	Driver name. The braces are required when using version 2.65 or earlier of the SQL Server ODBC driver.
Fallback (SQL Server 6.5 only)	yes	Fallback connection attempts are made if a connection to the primary server fails. Available only when connecting to an instance of SQL Server 6.5.
	no	Default. Fallback connection attempts are not made.
Language		 SQL Server language name. SQL Server can store messages for multiple languages in sysmessages. If connecting to a SQL Server with multiple languages, Language specifies which set of messages are used for the connection.
Network		Name of a Net-Library dynamic-link library. The name need not include the path and must not include the .dll file name extension, for example, Network=dbnmpntw.
QueryLog_On	yes	Enables logging of long-running queries.
	no	Default. Disables logging of long- running queries.
QueryLogFile		Full path and name of the file used to log long-running queries.
QueryLogTime		Digit character string specifying the threshold (in milliseconds) for logging long-running queries. Any query that does not get a response in the time specified is written to the long-running query log file.
QuotedId	yes	Default. Specifies that QUOTED_IDENTIFIERS is set ON for each connection, SQL Server uses the SQL-92 rules regarding the use of

		quotation marks in SQL statements. For more information, see Effects of SQL-92 Options.
	no	Specifies that QUOTED_IDENTIFIERS is set OFF for each connection. SQL Server then follows the legacy Transact- SQL rules regarding the use of quotation marks in SQL statements.
Regional	yes	Respect client workstation settings for region when converting date, time, and currency values to character strings. This setting should only be specified for applications that only display data, not for applications that process data.
	no	Default. Use ODBC-defined character formats for date, time, and money conversion.
Server		 Name of a server running SQL Server on the network. The value must be either the name of a server on the network, or the name of a SQL Server Client Network Utility advanced server entry. You can enter (local) as the server name on Windows NT 4.0 to connect to a copy of SQL Server running on the same computer. SQL Server 2000 supports multiple instances of SQL Server running on the same computer. To specify a named instance of SQL Server, the server name is specified as ServerName\InstanceName. For more information about server names, see <u>Managing Clients</u>.
StatsLog_On	yes	Enables driver performance logging.
	no	Default. Disables driver performance logging.

StatsLogFile		Full path and name of the file used to record SQL Server ODBC driver performance statistics.
Trusted_Connection	yes	Windows Authentication is enabled for the data source.
	no	Default. SQL Server Authentication is enabled for the data source. A SQL Server login and password must be specified for each connection.
UseProcForPrepare	0	Temporary stored procedures are not created on statement preparation.
(SQL Server 6.5 or earlier only)	1	Default. Temporary stored procedures are created for prepared SQL statements. The procedures are dropped when the connection is closed or lost.
	2	Temporary stored procedures are created for prepared SQL statements. The procedures are dropped when the cursor is closed.

Note Regional conversion settings apply to currency, numeric, date, and time data types. The conversion setting is only applicable to output conversion and is only visible when currency, numeric, date, or time values are converted to character strings.

The driver uses the locale registry settings for the current user. The driver does not honor the current thread's locale if the application sets it after connection by, for example, calling SetThreadLocale.

Altering the regional behavior of a data source can cause application failure. An application that parses date strings, and expects date strings to appear as defined by ODBC, could be adversely affected by altering this value.

SQLDescribeCol

For executed statements, the Microsoft® SQL Server[™] ODBC driver does not need to query the server to describe columns in a result set. In this case, **SQLDescribeCol** does not cause a server roundtrip. Like **SQLColAttribute** and **SQLNumResultCols**, calling **SQLDescribeCol** on prepared but not executed statements generates a server roundtrip.

When a Transact-SQL statement or statement batch returns multiple result row sets, it is possible for a column, referenced by ordinal, to originate in a separate table or to refer to an entirely different column in the result set. **SQLDescribeCol** should be called for each set. When the result set changes, the application should rebind data values prior to fetching row results. For more information about handling multiple result set returns, see <u>SQLMoreResults</u>.

Note When connected to an instance of SQL Server version 4.2*x*, **SQLDescribeCol** must create a result set to report column attributes. The SQL Server ODBC driver will append the clause WHERE 1 = 2 to prepared SELECT statements prior to execution. When connected to SQL Server 4.2*x*, **SQLDescribeCol** cannot return information about a result set that is generated by a procedure if that procedure has been prepared but not executed.

When connected to any later version of SQL Server, the SQL Server ODBC driver uses the SET FMTONLY statement to reduce server overhead when **SQLDescribeCol** is called for prepared but not executed statements.

For all versions, column attributes are reported for only the first result set when multiple result sets are generated by a prepared batch of SQL statements.

SQLDescribeParam

To describe the parameters of any SQL statement, the Microsoft® SQL Server[™] ODBC driver builds and executes a Transact-SQL SELECT statement when **SQLDescribeParam** is called on a prepared ODBC statement handle. The driver uses the SET FMTONLY statement when executing the query. The meta data of the result set determines the characteristics of the parameters in the prepared statement.

Consider this ODBC SQL statement:

INSERT INTO Shippers (*ShipperID***,** *CompanyName***,** *Phone***) VALUES (?, ?, ?)**

On a call to **SQLDescribeParam**, this ODBC SQL statement causes the driver to execute the following Transact-SQL statement:

SET FMTONLY ON SELECT *ShipperID*, *CompanyName*, *Phone* **FROM Shippers SET FMTONLY OFF**

SQLDescribeParam can, therefore, return any error code that **SQLExecute** or **SQLExecDirect** might return.

Further, the driver does not support calling **SQLDescribeParam** after **SQLExecDirect** for any Transact-SQL UPDATE or DELETE statements containing the FROM clause; for any ODBC or Transact-SQL statement depending on a subquery containing parameters; for ODBC SQL statements containing parameter markers in both expressions of a comparison, like, or quantified predicate; or queries where one of the parameters is a parameter to a function.

When processing a batch of Transact-SQL statements, the driver also does not support calling **SQLDescribeParam** for parameter markers in statements after the first statement in the batch.

When describing the parameters of prepared stored procedures,

SQLDescribeParam uses the system stored procedure **sp_sproc_columns** to retrieve parameter characteristics. **sp_sproc_columns** can report data for stored procedures within the current user database. Preparing a fully qualified stored procedure name allows **SQLDescribeParam** to execute across databases. For

example, the system stored procedure **sp_who** can be prepared and executed in any database as:

SQLPrepare(hstmt, "{call sp_who(?)}", SQL_NTS);

Executing **SQLDescribeParam** after successful preparation returns an empty row set when connected to any database but **master**. The same call, prepared as follows, causes **SQLDescribeParam** to succeed regardless of the current user database:

```
SQLPrepare(hstmt, "{call master..sp_who(?)}", SQL_NTS);
```

SQLDriverConnect

The Microsoft® SQL Server[™] ODBC driver and the ODBC driver manager recognize the following **SQLDriverConnect** connection string keywords.

Keyword	Description
Address	Network address of the server running an instance of SQL Server. Address is usually the network name of the server, but can be other names such as a pipe, or a TCP/IP port and socket address. For more information, see <u>Managing Clients</u> .
AnsiNPW	When yes , the driver uses ANSI-defined behaviors for handling NULL comparisons, character data padding, warnings, and NULL concatenation. When no , ANSI defined behaviors are not exposed. For more information about ANSI NPW behaviors, see <u>Effects of SQL-92 Options</u> .
APP	Name of the application calling SQLDriverConnect (optional). If specified, this value is stored in the master.dbo.sysprocesses column program_name and is returned by sp_who and the Transact-SQL APP_NAME function.
AttachDBFileName	Name of the primary file of an attachable database. Include the full path and escape any \ characters if using a C character string variable: AttachDBFileName=c:\\MyFolder\\MyDB.mdf This database is attached and becomes the default database for the connection. To use AttachDBFileName you must also specify the database name in either the SQLDriverConnect DATABASE parameter or the SQL_COPT_CURRENT_CATALOG connection attribute. If the database was previously attached, SQL Server will not reattach it; it will use the

	attached database as the default for the connection.
AutoTranslate	When yes , ANSI character strings sent between the client and server are translated by converting through Unicode to minimize problems in matching extended characters between the code pages on the client and the server:
	Client SQL_C_CHAR data sent to a SQL Server char , varchar , or text variable, parameter, or column is converted from character to Unicode using the client ANSI code page (ACP), then converted from Unicode to character using the ACP of the server.
	SQL Server char , varchar , or text data sent to a client SQL_C_CHAR variable is converted from character to Unicode using the server ACP, then converted from Unicode to character using the client ACP.
	These conversions are performed on the client by the SQL Server ODBC driver. This requires that the same ANSI code page (ACP) used on the server be available on the client.
	These settings have no effect on the conversions that occur for these transfers:
	Unicode SQL_C_WCHAR client data sent to char , varchar , or text on the server.
	char , varchar , or text server data sent to a Unicode SQL_C_WCHAR variable on the client.
	ANSI SQL_C_CHAR client data sent to Unicode nchar , nvarchar , or ntext on the server.
	Unicode char , varchar , or text server data sent to an ANSI SQL_C_CHAR variable on the client.
	When no , character translation is not performed.

	 The SQL Server ODBC driver does not translate client ANSI character SQL_C_CHAR data sent to char, varchar, or text variables, parameters, or columns on the server. No translation is performed on char, varchar, or text data sent from the server to SQL_C_CHAR variables on the client. If the client and SQL Server are using different ACPs, then extended characters can be misinterpreted.
DATABASE	 Name of the default SQL Server database for the connection. If Database is not specified, the default database defined for the login is used. The default database from the ODBC data source overrides the default database defined for the login. The database must be an existing database unless AttachDBFileName is also specified. If AttachDBFileName is also specified, the primary file it points to is attached and given the database name specified by DATABASE.
DRIVER	Name of the driver as returned by SQLDrivers . The keyword value for the SQL Server ODBC driver is " {SQL Server}". The braces are required when using version 2.65 or earlier of the SQL Server ODBC driver. The SERVER keyword is required if DRIVER is specified and <i>DriverCompletion</i> is set to SQL_DRIVER_NOPROMPT.
DSN	Name of an existing ODBC user or system data source.
Fallback (SQL Server 6.5 only)	When yes , instructs the driver to attempt connection to a fallback server if connection to a primary server fails. The login time-out (set with ODBC SQLSetConnectAttr , attribute SQL_ATTR_LOGIN_TIMEOUT) must be set for fallback to occur. When no , no attempt at a fallback connection is made. This option applies only to

	standby servers. It does not apply to a virtual server in a cluster/failover configuration.
FILEDSN	Name of an existing ODBC file data source.
LANGUAGE	 SQL Server language name (optional). SQL Server can store messages for multiple languages in sysmessages. If connecting to a SQL Server with multiple languages, Language specifies which set of messages are used for the connection.
Network	Name of a network library dynamic-link library. The name need not include the path and must not include the .dll file name extension, for example, Network=dbnmpntw.
PWD	The password for the SQL Server login account specified in the UID parameter. PWD need not be specified if the login has a NULL password or when using Windows Authentication (Trusted_Connection = yes).
SAVEFILE	Name of an ODBC data source file into which the attributes of the current connection are saved if the connection is successful.
SERVER	Name of a server running SQL Server on the network. The value must be either the name of a server on the network, or the name of a SQL Server Client Network Utility advanced server entry. You can enter (local) as the server name on Microsoft Windows® NT 4.0 to connect to a copy of SQL Server running on the same computer. SQL Server 2000 supports multiple instances of SQL Server running on the same computer. To specify a named instance of SQL Server, the server name is specified as ServerName\InstanceName. For more information about server names, see <u>Managing Clients</u> .
QueryLogFile	Full path and file name of a file to use to log data on long-running queries.
QueryLog_On	When yes , logging long-running query data is enabled on the connection. When no , long-running

	query data is not logged.
QueryLogTime	Digit character string specifying the threshold (in milliseconds) for logging long-running queries. Any query that does not get a response in the time specified is written to the long-running query log file.
QuotedID	When yes , QUOTED_IDENTIFIERS is set ON for the connection, SQL Server uses the SQL-92 rules regarding the use of quotation marks in SQL statements. When no , QUOTED_IDENTIFIERS is set OFF for the connection. SQL Server then follows the legacy Transact-SQL rules regarding the use of quotation marks in SQL statements. For more information, see <u>Effects of SQL-92 Options</u> .
Regional	When yes , the SQL Server ODBC driver uses client settings when converting currency, date, and time data to character data. The conversion is one way only; the driver does not recognize non-ODBC standard formats for date strings or currency values within; for example, a parameter used in an INSERT or UPDATE statement. When no , the driver uses ODBC standard strings to represent currency, date, and time data that is converted to string data.
StatsLogFile	Full path and file name of a file used to record SQL Server ODBC driver performance statistics.
StatsLog_On	When yes , enables the capture of SQL Server ODBC driver performance data. When no , SQL Server ODBC driver performance data is not available on the connection.
Trusted_Connection	When yes , instructs the SQL Server ODBC driver to use Windows Authentication Mode for login validation. The UID and PWD keywords are optional. When no , instructs the SQL Server ODBC driver to use a SQL Server username and password for login validation. The UID and PWD keywords must be specified.

UID	A valid SQL Server login account. UID need not be specified when using Windows Authentication.
UseProcForPrepare (SQL Server 6.5 and earlier only)	When 1 , instructs the SQL Server ODBC driver to create temporary stored procedures when statements are prepared with SQLPrepare . The temporary stored procedures are not dropped until the connection is broken.
	When 2 , the SQL Server ODBC driver creates temporary stored procedures for SQLPrepare , but only one procedure is created per statement handle and the procedure is dropped when the statement handle becomes invalid or a new SQL statement is prepared. When 0 , the SQL Server ODBC driver does not create temporary stored procedures for SQLPrepare .
WSID	Workstation ID. Typically, this is the network name of the computer on which the application resides (optional). If specified, this value is stored in the master.dbo.sysprocesses column hostname and is returned by sp_who and the Transact-SQL HOST_NAME function.

Note Regional conversion settings apply to currency, numeric, date, and time data types. The conversion setting is only applicable to output conversion and is only visible when currency, numeric, date, or time values are converted to character strings.

The driver uses the locale registry settings for the current user. The driver does not honor the current thread's locale if the application sets it after connection by, for example, calling SetThreadLocale.

Altering the regional behavior of a data source can cause application failure. An application that parses date strings, and expects date strings to appear as defined by ODBC, could be adversely affected by altering this value.

The SQL Server ODBC driver defines connection attributes that either replace or

enhance connection-string keywords. Several connection-string keywords have default values specified by the SQL Server ODBC driver. For more information about SQL Server connection attributes and driver default behaviors, see <u>SQLSetConnectAttr</u>.

When the **SQLDriverConnect** *DriverCompletion* parameter value is SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, or SQL_DRIVER_COMPLETE_REQUIRED, the SQL Server ODBC driver retrieves keyword values from the displayed dialog box. If the keyword value is passed in the connection string and the user does not alter the value for the keyword in the dialog box, the SQL Server ODBC driver uses the value from the connection string. If the value is not set in the connection string and the user makes no assignment in the dialog box, the driver uses the default.

SQLDriverConnect must be given a valid *WindowHandle* when any *DriverCompletion* value requires (or could require) the display of the driver's connection dialog box. An invalid handle returns SQL_ERROR.

Specify either the DRIVER or DSN keywords. ODBC states that a driver uses the leftmost of these two keywords and ignores the other if both are specified. If DRIVER is specified, or is the leftmost of the two, and the **SQLDriverConnect** *DriverCompletion* parameter value is SQL_DRIVER_NOPROMPT, the SERVER keyword and an appropriate value are required.

When SQL_DRIVER_NOPROMPT is specified, user authentication keywords must be present with values. The driver ensures that either the string "Trusted_Connection=yes" or both the UID and PWD keywords are present.

If the *DriverCompletion* parameter value is SQL_DRIVER_NOPROMPT or SQL_DRIVER_COMPLETE_REQUIRED and the language or database comes from the connection string and either is invalid, **SQLDriverConnect** returns SQL_ERROR.

If the *DriverCompletion* parameter value is SQL_DRIVER_NOPROMPT or SQL_DRIVER_COMPLETE_REQUIRED and the language or database comes from the ODBC data source definitions and either is invalid,

SQLDriverConnect uses the default language or database for the specified user ID and returns SQL_SUCCESS_WITH_INFO.

If the *DriverCompletion* parameter value is SQL_DRIVER_COMPLETE or SQL_DRIVER_PROMPT and if the language or database is invalid,

SQLDriverConnect redisplays the dialog box.

Examples

The following call illustrates the least amount of data required for **SQLDriverConnect**:

SQLDriverConnect(hdbc, hwnd, (SQLTCHAR*) "DRIVER={SQL Server};" SQL_NTS, szOutConn, MAX_CONN_OUT, cbOutConn, SQL_DRIVER_COMPLETE);

The following connection strings illustrate minimum required data when the *DriverCompletion* parameter value is SQL_DRIVER_NOPROMPT:

"DSN=Human Resources;UID=Smith;PWD=Sesame"

"DSN=Human Resources;Trusted_Connection=yes"

"FILEDSN=HR_FDSN;UID=Smith;PWD=Sesame"

"FILEDSN=HR_FDSN;Trusted_Connection=yes"

"DRIVER={SQL Server};SERVER=hrserver;UID=Smith;PWD=Sesa

"DRIVER={SQL Server};SERVER=hrserver;Trusted_Connection=yes

See Also

<u>SET ANSI_NULLS</u>

SET ANSI_PADDING

<u>SET ANSI_WARNINGS</u>

SQLDrivers

The ODBC Driver Manager returns all ODBC 3.0-defined **SQLDrivers** attribute specification strings. For more information about attribute string and value definition, see the ODBC 3.0 documentation.

SQLEndTran

By default, the Microsoft® SQL Server[™] ODBC driver closes a statement's associated cursor when **SQLEndTran** commits or rolls back an operation. Server cursors are closed unless they are static.

SQLFetchScroll

SQLFetchScroll returns one row set of data to the application. The size of the row set is set using **SQLSetStmtAttr**. The Microsoft® SQL Server[™] ODBC driver supports all defined fetch instructions (for example, SQL_FETCH_RELATIVE) with the following limitations:

- If a forward-only cursor is defined for the statement, SQL_FETCH_NEXT is required and attempts to fetch in any other fashion will result in an error return.
- SQL_FETCH_BOOKMARK is supported for static and keyset-driven cursors only.

SQLForeignKeys

SQLForeignKeys uses the catalog stored procedure **sp_fkeys** to report foreign keys referencing a table's primary key or columns in a table that reference the primary key columns of other tables.

The following table shows the **SQLForeignKeys** parameter mapping for **sp_fkeys** stored procedure execution.

SQLForeignKeys parameter name	sp_fkeys parameter name
PKTableCatalog	pktable_qualifier
PKTableSchema	pktable_owner
PKTableName	pktable_name
FKTableCatalog	fktable_qualifier
FKTableSchema	fktable_owner
FKTableName	fktable_name

Microsoft® SQL Server[™] supports cascading updates and deletes through the foreign key constraint mechanism. SQL Server returns SQL_CASCADE for UPDATE_RULE and/or DELETE_RULE columns if CASCADE option is specified on the ON UPDATE and/or ON DELETE clause of the FOREIGN KEY constraints. SQL Server returns SQL_NO_ACTION for UPDATE_RULE and/or DELETE_RULE columns if NO ACTION option is specified on the ON UPDATE and/or ON DELETE clause of the FOREIGN KEY constraints.

When invalid values are present in any **SQLForeignKeys** parameter, **SQLForeignKeys** returns SQL_SUCCESS on execution. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLForeignKeys can be executed on a static server cursor. An attempt to execute **SQLForeignKeys** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

The SQL Server ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *FKCatalogName* and *PKCatalogName* parameters: *Linked_Server_Name.Catalog_Name*.

See Also

<u>sp_fkeys</u>

SQLFreeHandle

In manual-commit mode, calling **SQLFreeHandle** on a statement handle with an open transaction causes a rollback of pending changes to the database. **SQLFreeHandle** of a statement handle always closes any open cursors and discards pending results, freeing all resources associated with the statement handle.

SQLFreeStmt

SQLFreeStmt is not recommended in ODBC 3.0 and later. The Microsoft® SQL Server[™] ODBC driver supports all defined *Option* values for **SQLFreeStmt**. However, **SQLCloseCursor**, **SQLBindParameter**, **SQLBindCol**, **SQLSetDescField**, and **SQLFreeHandle** replace or duplicate the function of **SQLFreeStmt** and should be used instead.

See Also

SQLBindCol

<u>SQLCloseCursor</u>

SQLBindParameter

SQLGetConnectAttr

The Microsoft® SQL Server[™] ODBC driver defines driver-specific connection attributes. Some of the attributes are available to **SQLGetConnectAttr**, and the function is used to report their current settings. The values reported for these attributes are not guaranteed until after a connection has been made or the attribute has been set using **SQLSetConnectAttr**.

SQL_COPT_SS_ANSI_NPW

SQL_COPT_SS_ANSI_NPW enables or disables the use of ANSI handling of NULL comparisons, character data type padding, warning levels, and NULL concatenation. For more information, see SET ANSI_NULLS, SET ANSI_PADDING, SET ANSI_WARNINGS, and SET CONCAT_NULL_YIELDS_NULL.

Value	Description
SQL_AD_ON	Default. The connection uses ANSI default behavior handling NULL comparisons, padding, warnings, and NULL concatenations.
SQL_AD_OFF	The connection uses SQL Server defined handling of NULL comparisons, character data type padding, warnings, and NULL concatenations.

SQL_COPT_SS_CONNECTION_DEAD

SQL_COPT_SS_CONNECTION_DEAD reports the alive or dead state of a connection to a server. The driver queries the Net-Library for the current state of the connection.

Value	Description	
SQL_CD_TRUE	The connection to the server has been lost.	
SQL_CD_FALSE	The connection is open and available for statement	
	processing.	

SQL_COPT_SS_PERF_DATA

SQL_COPT_SS_PERF_DATA returns a pointer to a SQLPERF structure containing the current driver performance statistics. **SQLGetConnectAttr** will return NULL if performance logging is not enabled. The statistics in the SQLPERF structure are not dynamically updated by the driver. Call **SQLGetConnectAttr** each time the performance statistics need to be refreshed. For more information about performance logging, see <u>SQLSetConnectAttr</u>.

Value	Description	
NULL	Performance logging is not enabled.	
Any other value	A pointer to a SQLPERF structure.	

SQL_COPT_SS_PERF_QUERY

SQL_COPT_SS_PERF_QUERY returns TRUE if logging of long running queries is enabled. The request returns FALSE if query logging is not active.

SQL_COPT_SS_PRESERVE_CURSORS

SQL_COPT_SS_PRESERVE_CURSORS defines the behavior of cursors when manual-commit mode is used. The behavior is exposed as transactions and are either committed or rolled back using **SQLEndTran**.

Value	Description	
SQL_PC_OFF	Default. Cursors are closed on SQLEndTran .	
SQL_PC_ON	Cursors remain open after the call to SQLEndTran .	

SQL_COPT_SS_QUOTED_IDENT

SQL_COPT_SS_QUOTED_IDENT allows quoted identifiers in ODBC and Transact-SQL statements submitted on the connection. By supplying quoted identifiers, the SQL Server ODBC driver allows otherwise invalid object names such as "My Table," which contains a space character in the identifier.

Value	Description
SQL_QI_OFF	The SQL Server connection does not allow quoted identifiers in submitted Transact-SQL.
SQL_QI_ON	Default. The connection allows quoted identifiers in Transact-SQL submitted.

SQL_COPT_SS_TRANSLATE

SQL_COPT_SS_TRANSLATE controls character translation as MBCS data is exchanged. The attribute affects only data stored in SQL Server **char**, **varchar**, and **text** columns.

Value	Description
SQL_XL_OFF	The SQL Server ODBC driver does not translate
	characters from one code page to another in character
	data exchanged between the client and the server.
SQL_XL_ON	Default. The SQL Server ODBC driver translates
	characters from one code page to another in character
	data exchanged between the client and the server. The
	driver automatically configures the character
	translation, determining the code page installed on the
	server and that in use by the client.

SQL_COPT_SS_USE_PROC_FOR_PREP

This option is valid only when connected to SQL Server version 6.5 or earlier. SQL_COPT_SS_USE_PROC_FOR_PREP defines the use of temporary stored procedures when ODBC and Transact-SQL statements are prepared for execution. For more information about prepared statement execution, see <u>SQLPrepare</u>.

Value	Description	
SQL_UP_OFF	The driver does not generate stored procedures when	
	the application prepares statements.	

Default. The driver generates a temporary stored procedure when a statement is prepared. The stored procedure is dropped when the application disconnects from the server.	
P The driver generates a temporary stored procedure when a statement is prepared. The stored procedure is dropped when the statement handle is freed.	

SQL_COPT_SS_USER_DATA

SQL_COPT_SS_USER_DATA retrieves the user-data pointer. User data is stored in client-owned memory and recorded per connection. If the user-data pointer has not been set, SQL_UD_NOTSET, a NULL pointer, is returned.

Value	Description
SQL_UD_NOTSET	No user-data pointer is set.
Any other value	A pointer to the user data.

See Also

Delimited Identifiers

SET ANSI_NULLS

SET ANSI_PADDING

SET ANSI_WARNINGS

SQLEndTran

SQLGetCursorName

If the application does not specify a cursor name, the Microsoft® SQL ServerTM ODBC driver generates one for the application upon cursor generation. The application can use **SQLGetCursorName** to retrieve the driver-defined cursor name for positioned UPDATE and DELETE statements. The application does not need to call **SQLSetCursorName** to take advantage of positioned data manipulation statements.

SQLGetData

SQLGetData is used to retrieve result set data without binding column values. **SQLGetData** can be called successively on the same column to retrieve large amounts of data from a column with a **text**, **ntext**, or **image** data type.

There is no requirement that an application bind variables to fetch result set data. The data of any column can be retrieved from the Microsoft® SQL Server[™] ODBC driver by using **SQLGetData**.

The SQL Server ODBC driver does not support using **SQLGetData** to retrieve data in random column order. All unbound columns processed with **SQLGetData** must have higher column ordinals than the bound columns in the result set. The application must process data from the lowest unbound ordinal column value to the highest. Attempting to retrieve data from a lower ordinally numbered column results in an error. If the application is using server cursors to report result set rows, the application can refetch the current row and then fetch the value of a column. If a statement is executed on the default read-only, forward-only cursor, you must re-execute the statement to back up **SQLGetData**.

The SQL Server ODBC driver accurately reports the length of **text**, **ntext**, and **image** data retrieved using **SQLGetData**. The application can make good use of the *StrLen_or_IndPtr* parameter return to retrieve long data rapidly.

Examples

```
SQLHDBC hDbc = NULL;
SQLHSTMT hStmt = NULL;
long lEmpID;
PBYTE pPicture;
SQLINTEGER pIndicators[2];
```

// Get an environment, connection, and so on.

```
// Get a statement handle and execute a command.
SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);
```

```
if (SQLExecDirect(hStmt,
  (SQLCHAR*) "SELECT EmployeeID, Photo FROM Employees",
  SQL_NTS) == SQL_ERROR)
  {
  // Handle error and return.
  }
// Retrieve data from row set.
SQLBindCol(hStmt, 1, SQL_C_LONG, (SQLPOINTER) & lEmpID, s
  &pIndicators[0]);
while (SQLFetch(hStmt) == SQL_SUCCESS)
  printf("EmployeeID: %d\n", lEmpID);
  // Call SQLGetData to determine the amount of data that's waiting.
  if (SQLGetData(hStmt, 2, SQL_C_BINARY, pPicture, 0, &pIndicat
    == SQL_SUCCESS_WITH_INFO)
    {
    printf("Photo size: %ld\n\n", pIndicators[1]);
    // Get all the data at once.
    pPicture = new BYTE[pIndicators[1]];
    if (SQLGetData(hStmt, 2, SQL_C_DEFAULT, pPicture,
      pIndicators[1], &pIndicators[1]) != SQL SUCCESS)
      // Handle error and continue.
```

```
delete [] pPicture;
```

```
}
else
{
// Handle error on attempt to get data length.
}
```

SQLGetDescField

The Microsoft® SQL Server[™] ODBC driver exposes driver-specific descriptor fields for the implementation row descriptor (IRD) only. Within the IRD, SQL Server descriptor fields are referenced through driver-specific column attributes. For information about a complete list of available driver-specific descriptor fields, see <u>SQLColAttribute</u>.

Descriptor fields that contain column identifier strings are often zero length strings. For a description of the behavior of ODBC descriptor fields that contain column identifier strings, see <u>SQLColAttribute</u>.

All SQL Server-specific descriptor field values are read-only.

Like attributes retrieved with **SQLColAttribute**, descriptor fields that report row-level attributes (such as SQL_CA_SS_COMPUTE_ID) are reported for all columns in the result set.

Example

```
...
typedef struct tagCOMPUTEBYLIST
{
SQLSMALLINT nBys;
SQLSMALLINT aByList[1];
} COMPUTEBYLIST;
typedef COMPUTEBYLIST* PCOMPUTEBYLIST;
```

```
SQLHDESC hIRD;
SQLINTEGER cbIRD;
SQLINTEGER nSet = 0;
```

// . . .

// Execute a statement that contains a COMPUTE clause,

// then get the descriptor handle of the IRD and

// get some IRD values.

```
SQLGetStmtAttr(g_hStmt, SQL_ATTR_IMP_ROW_DESC,
  (SQLPOINTER) & hIRD, sizeof(SQLHDESC), & cbIRD);
// For statement-wide column attributes, any
// descriptor record will do. You know that 1 exists,
// so use it.
SQLGetDescField(hIRD, 1, SQL_CA_SS_NUM_COMPUTES,
  (SQLPOINTER) &nComputes, SQL_IS_INTEGER, &cbIRD);
if (nSet == 0)
  ł
                  nOrderID;
  SQLINTEGER
  printf("Normal result set.\n");
  for (nCol = 0; nCol < nCols; nCol++)
    ł
    SQLGetDescField(hIRD, nCol+1,
      SQL_CA_SS_COLUMN_ORDER,
      (SQLPOINTER) &nOrderID, SQL_IS_INTEGER,
      &cbIRD);
    if (nOrderID != 0)
      ł
      printf("Col in ORDER BY, pos: %ld",
        nOrderID);
      }
      printf("\n");
    }
```

```
printf("\n");
```

```
}
else
  ł
  PCOMPUTEBYLIST pByList;
  SQLSMALLINT
                    nBy;
  SQLINTEGER
                   nColID;
  printf("Computed result set number: %lu\n",
    nSet);
  SQLGetDescField(hIRD, 1, SQL_CA_SS_COMPUTE_BYLIST,
    (SQLPOINTER) &pByList, SQL_IS_INTEGER,
    &cbIRD);
  if (pByList != NULL)
    printf("Clause ordered by columns: ");
    for (nBy = 0; nBy < pByList->nBys; )
      {
      printf("%u", pByList->aByList[nBy]);
      nBy++;
      if (nBy == pByList->nBys)
         {
         printf("\n");
         }
      else
         {
         printf(", ");
         ł
      }
    }
  else
```

```
{
    printf("Compute clause set not ordered.\n");
  for (nCol = 0; nCol < nCols; nCol++)</pre>
    {
    SQLGetDescField(hIRD, nCol+1,
      SQL_CA_SS_COLUMN_ID, (SQLPOINTER) &nColID,
      SQL_IS_INTEGER, &cbIRD);
    printf("ColumnID: %lu, nColID);
    }
  printf("\n");
  }
if (SQLMoreResults(g_hStmt) == SQL_SUCCESS)
  ł
  // Determine the result set indicator.
  SQLGetDescField(hIRD, 1, SQL_CA_SS_COMPUTE_ID,
    (SQLPOINTER) &nSet, SQL_IS_INTEGER, &cbIRD);
  }
```

```
// and carry on...
```

SQLGetDiagField

The Microsoft® SQL Server[™] ODBC driver specifies the following additional diagnostics fields for **SQLGetDiagField**. These fields support rich error reporting for SQL Server applications and are available in all diagnostics records generated on connected ODBC connection handles and ODBC statement handles. The fields are defined in Odbcss.h.

Diagnostics record field	Description
SQL_DIAG_SS_LINE	Reports the line number of a stored procedure generating an error. The value of SQL_DIAG_SS_LINE is meaningful only if SQL_DIAG_SS_PROCNAME returns a value. The value is returned as an unsigned, 16-bit integer.
SQL_DIAG_SS_MSGSTATE	The state of an error message. For information about the error message state, see <u>RAISERROR</u> . The value is returned as a signed, 32-bit integer.
SQL_DIAG_SS_PROCNAME	Name of the stored procedure generating an error, if appropriate. The value is returned as a character string. The length of the string (in characters) depends on the version of the SQL Server. It can be determined by calling SQLGetInfo requesting the value for SQL_MAX_PROCEDURE_NAME_LEN.
SQL_DIAG_SS_SEVERITY	The severity level of the associated error message. The value is returned as a signed, 32-bit integer.
SQL_DIAG_SS_SRVNAME	The name of the server on which the error occurred. The value is returned as a character string. The length of the string (in characters) is defined by the SQL_MAX_SQLSERVERNAME macro in

Odbcss.h.

SQL Server-specific diagnostic fields that contain character data, SQL_DIAG_SS_PROCNAME and SQL_DIAG_SS_SRVNAME, return that data to the client as null terminated, ANSI, or Unicode strings. If necessary, the count of characters should be adjusted by the character width. Alternately, a portable C data type such as TCHAR or SQLTCHAR can be used to ensure correct program variable length.

The SQL Server ODBC driver reports the following additional dynamic function codes that identify the last attempted SQL Server statement. The dynamic function code is returned in the header (record 0) of the diagnostics record set and is therefore available on every execution (successful or not).

Dynamic function code	Source
SQL_DIAG_DFC_SS_ALTER_DATABASE	ALTER DATABASE
	statement
SQL_DIAG_DFC_SS_CHECKPOINT	CHECKPOINT
	statement
SQL_DIAG_DFC_SS_CONDITION	Error arose in the
	WHERE or HAVING
	clauses of a statement.
SQL_DIAG_DFC_SS_CREATE_DATABASE	CREATE DATABASE
	statement
SQL_DIAG_DFC_SS_CREATE_DEFAULT	CREATE DEFAULT
	statement
SQL_DIAG_DFC_SS_CREATE_PROCEDURE	CREATE
	PROCEDURE
	statement
SQL_DIAG_DFC_SS_CREATE_RULE	CREATE RULE
	statement
SQL_DIAG_DFC_SS_CREATE_TRIGGER	CREATE TRIGGER
	statement
SQL_DIAG_DFC_SS_CURSOR_DECLARE	DECLARE CURSOR
	statement
SQL_DIAG_DFC_SS_CURSOR_OPEN	OPEN statement
SQL_DIAG_DFC_SS_CURSOR_FETCH	FETCH statement

	1
SQL_DIAG_DFC_SS_CURSOR_CLOSE	CLOSE statement
SQL_DIAG_DFC_SS_DEALLOCATE_CURSOR	DEALLOCATE
	statement
SQL_DIAG_DFC_SS_DBCC	DBCC statement
SQL_DIAG_DFC_SS_DENY	DENY statement
SQL_DIAG_DFC_SS_DISK	DISK INIT statement
SQL_DIAG_DFC_SS_DROP_DATABASE	DROP DATABASE
	statement
SQL_DIAG_DFC_SS_DROP_DEFAULT	DROP DEFAULT
	statement
SQL_DIAG_DFC_SS_DROP_PROCEDURE	DROP PROCEDURE
	statement
SQL_DIAG_DFC_SS_DROP_RULE	DROP RULE statement
SQL_DIAG_DFC_SS_DROP_TRIGGER	DROP TRIGGER
	statement
SQL_DIAG_DFC_SS_DUMP_DATABASE	BACKUP or DUMP
	DATABASE statement
SQL_DIAG_DFC_SS_DUMP_TABLE	DUMP TABLE
	statement
SQL_DIAG_DFC_SS_DUMP_TRANSACTION	BACKUP or DUMP
	TRANSACTION
	statement. Also
	returned for a
	CHECKPOINT
	statement if the trunc .
	log on chkpt. database
SOL DIAC DEC SS COTO	option is on. GOTO control-of-flow
SQL_DIAG_DFC_SS_GOTO	GOTO control-of-flow statement
SOL DIAC DEC SS INSEDT DIUV	
SQL_DIAG_DFC_SS_INSERT_BULK	INSERT BULK statement
SOL DIAC DEC SS KULL	
SQL_DIAG_DFC_SS_KILL	KILL statement
SQL_DIAG_DFC_SS_LOAD_DATABASE	LOAD or RESTORE
	DATABASE statement

SQL_DIAG_DFC_SS_LOAD_HEADERONLY	LOAD or RESTORE HEADERONLY statement
SQL_DIAG_DFC_SS_LOAD_TABLE	LOAD TABLE statement
SQL_DIAG_DFC_SS_LOAD_TRANSACTION	LOAD or RESTORE TRANSACTION statement
SQL_DIAG_DFC_SS_PRINT	PRINT statement
SQL_DIAG_DFC_SS_RAISERROR	RAISERROR statement
SQL_DIAG_DFC_SS_READTEXT	READTEXT statement
SQL_DIAG_DFC_SS_RECONFIGURE	RECONFIGURE statement
SQL_DIAG_DFC_SS_RETURN	RETURN control-of- flow statement
SQL_DIAG_DFC_SS_SELECT_INTO	SELECT INTO statement
SQL_DIAG_DFC_SS_SET	SET statement (generic, all options)
SQL_DIAG_DFC_SS_SET_IDENTITY_INSERT	SET IDENTITY_INSERT statement
SQL_DIAG_DFC_SS_SET_ROW_COUNT	SET ROWCOUNT statement
SQL_DIAG_DFC_SS_SET_STATISTICS	SET STATISTICS IO or SET STATISTICS TIME statements
SQL_DIAG_DFC_SS_SET_TEXTSIZE	SET TEXTSIZE statement
SQL_DIAG_DFC_SS_SETUSER	SETUSER statement
SQL_DIAG_DFC_SS_SET_XCTLVL	SET TRANSACTION ISOLATION LEVEL statement
SQL_DIAG_DFC_SS_SHUTDOWN	SHUTDOWN

	statement
SQL_DIAG_DFC_SS_TRANS_BEGIN	BEGIN TRAN
	statement
SQL_DIAG_DFC_SS_TRANS_COMMIT	COMMIT TRAN
	statement
SQL_DIAG_DFC_SS_TRANS_PREPARE	Prepare to commit a
	distributed transaction
SQL_DIAG_DFC_SS_TRANS_ROLLBACK	ROLLBACK TRAN
	statement
SQL_DIAG_DFC_SS_TRANS_SAVE	SAVE TRAN statement
SQL_DIAG_DFC_SS_TRUNCATE_TABLE	TRUNCATE TABLE
	statement
SQL_DIAG_DFC_SS_UPDATE_STATISTICS	UPDATE STATISTICS
	statement
SQL_DIAG_DFC_SS_UPDATETEXT	UPDATETEXT
	statement
SQL_DIAG_DFC_SS_USE	USE statement
SQL_DIAG_DFC_SS_WAITFOR	WAITFOR control-of-
	flow statement
SQL_DIAG_DFC_SS_WRITETEXT	WRITETEXT
	statement

See Also

<u>SQLGetInfo</u>

SQLGetFunctions

The Microsoft® SQL Server[™] ODBC driver interface complies with the three defined compliance levels for ODBC (ISO, X/Open, and ODBC level 2). **SQLGetFunctions** reports that all ODBC functions are supported for the SQL Server ODBC driver.

SQLGetInfo

The table shows the values returned by **SQLGetInfo** (may vary based on the version number of the connected server).

fInfoType	rgbInfoValue
SQL_ACCESSIBLE_PROCEDURES	"Y"
SQL_ACCESSIBLE_TABLES	"Y"
SQL_ACTIVE_CONNECTIONS	The number of connecti Microsoft® SQL Server for this SQLGetInfo re
SQL_ACTIVE_ENVIRONMENTS	The number of environr driver. The driver return request.
SQL_ACTIVE_STATEMENTS	The driver returns 1 for The number of statemer application is not limite default execution on a s block execution on any
SQL_ALTER_DOMAIN	FALSE
SQL_ALTER_TABLE	SQL_AT_ADD_COLU SQL_AT_ADD_COLU SQL_AT_ADD_COLU SQL_AT_ADD_CONS' SQL_AT_ADD_TABLI SQL_AT_CONSTRAIN SQL_AT_DROP_COLU
SQL_SQL_CONFORMANCE	SQL_SC_SQL92_ENT
SQL_DATETIME_LITERALS	FALSE
SQL_ASYNC_MODE	SQL_AM_STATEMEN
SQL_BATCH_ROW_COUNT	SQL_BRC_EXPLICIT
SQL_BATCH_SUPPORT	SQL_BS_ROW_COUN SQL_BS_ROW_COUN SQL_BS_SELECT_EX

	SQL_BS_SELECT_PR
SQL_BOOKMARK_PERSISTENCE	SQL_BP_DELETE
	SQL_BP_SCROLL
	SQL_BP_UPDATE
SQL_CATALOG_LOCATION	SQL_CL_START
SQL_CATALOG_NAME	"Y"
SQL_CATALOG_NAME_SEPARATOR	"."
SQL_CATALOG_TERM	"database"
SQL_CATALOG_USAGE	SQL_CU_DML_STATI
	SQL_CU_PROCEDUR
	SQL_CU_TABLE_DEF
SQL_COLLATION_SEQ	The currently assigned of
	connection and server.
SQL_COLUMN_ALIAS	"Y"
SQL_CONCAT_NULL_BEHAVIOR	SQL_CB_NULL
	SQL_CB_NON_NULL
	6.5 or earlier server, or i
	connected to an instance
SQL_CONVERT_BIGINT	No support for conversi
	SQL_BIGINT data type
	driver supports the SQL data type as ODBC type
	SQL_CONVERT_DEC
SQL_CONVERT_BINARY	SQL_CVT_CHAR
SQL_CONVERI_DINART	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLIN
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_TINYINT
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WVARCH/
SQL_CONVERT_BIT	SQL_CVT_CHAR

	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_WCHAR
	SQL_CVT_WVARCHA
SQL_CONVERT_CHAR	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_LONGVAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_TIMESTAN
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WLONGVA
	SQL_CVT_WVARCHA
SQL_CONVERT_DATE	No support for conversion
	SQL_TYPE_DATE data
	ODBC driver supports t
	data type as ODBC type
	SQL_TYPE_TIMESTA
	SQL_CONVERT_TIMI

SQL_CONVERT_DECIMAL	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLIN SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_VARCHAR SQL_CVT_BINARY SQL_CVT_BINARY SQL_CVT_BIT SQL_CVT_BIT SQL_CVT_TINYINT SQL_CVT_WCHAR SQL_CVT_WVARCHA
SQL_CONVERT_DOUBLE	No support for conversi SQL_DOUBLE data tyj ODBC driver supports t SQL_DOUBLE data tyj SQL_CONVERT_FLO.
SQL_CONVERT_FLOAT	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLIN SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_VARCHAR SQL_CVT_BIT SQL_CVT_TINYINT SQL_CVT_WCHAR SQL_CVT_WCHAR
SQL_CONVERT_FUNCTIONS	SQL_FN_CVT_CONV SQL_FN_CVT_CAST
SQL_CONVERT_INTEGER	SQL_CVT_CHAR SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER

	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_WCHAR
	SQL_CVT_WVARCHA
SQL_CONVERT_INTERVAL_YEAR_MONTH	No support for conversion
SQL_CONVERT_INTERVAL_DAY_TIME	No support for conversion
SQL_CONVERT_LONGVARBINARY	SQL_CVT_BINARY
	SQL_CVT_LONGVAR
	SQL_CVT_VARBINAI
SQL_CONVERT_LONGVARCHAR	SQL_CVT_CHAR
	SQL_CVT_VARCHAR
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WLONGVA
	SQL_CVT_WVARCHA
SQL_CONVERT_NUMERIC	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_WCHAR
	SQL_CVT_WVARCHA
SQL_CONVERT_REAL	SQL_CVT_CHAR

	SQL_CVT_NUMERIC SQL_CVT_DECIMAL SQL_CVT_INTEGER SQL_CVT_SMALLIN SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_VARCHAR SQL_CVT_BIT SQL_CVT_BIT SQL_CVT_TINYINT SQL_CVT_WCHAR SQL_CVT_WCHAR
SQL_CONVERT_SMALLINT	SQL_CVT_CHARSQL_CVT_NUMERICSQL_CVT_DECIMALSQL_CVT_INTEGERSQL_CVT_SMALLINTSQL_CVT_FLOATSQL_CVT_REALSQL_CVT_REALSQL_CVT_BINARYSQL_CVT_BINARYSQL_CVT_BITSQL_CVT_BITSQL_CVT_TINYINTSQL_CVT_WCHARSQL_CVT_WCHARSQL_CVT_WVARCHAR
SQL_CONVERT_TIME	No support for conversi SQL_TYPE_TIME data ODBC driver supports t data type as ODBC type SQL_TYPE_TIMESTA SQL_CONVERT_TIMI
SQL_CONVERT_TIMESTAMP	SQL_CVT_CHAR SQL_CVT_VARCHAR SQL_CVT_BINARY SQL_CVT_VARBINAI SQL_CVT_TIMESTAN SQL_CVT_WCHAR

	SQL_CVT_WVARCH4
SQL_CONVERT_TINYINT	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_WCHAR
	SQL_CVT_WVARCH/
SQL_CONVERT_VARBINARY	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_VARCHAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_TINYINT
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WVARCHA
SQL_CONVERT_VARCHAR	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_LONGVAR
	SQL_CVT_BINARY

	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_TIMESTAN
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WLONGV
	SQL_CVT_WVARCH4
SQL_CONVERT_WCHAR	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLINT
	SQL_CVT_FLOAT
	SQL_CVT_REAL
	SQL_CVT_VARCHAR
	SQL_CVT_LONGVAR
	SQL_CVT_BINARY
	SQL_CVT_VARBINAI
	SQL_CVT_BIT
	SQL_CVT_TINYINT
	SQL_CVT_TIMESTAN
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WLONGVA
	SQL_CVT_WVARCHA
SQL CONVERT WLONGVARCHAR	SQL_CVT_CHAR
	SQL_CVT_VARCHAR
	SQL_CVT_LONGVAR
	SQL_CVT_WCHAR
	SQL_CVT_WLONGV
	SQL_CVT_WVARCH
SQL_CONVERT_WVARCHAR	SQL_CVT_CHAR
	SQL_CVT_NUMERIC
	SQL_CVT_DECIMAL
	SQL_CVT_INTEGER
	SQL_CVT_SMALLIN

	SQL_CVT_FLOAT SQL_CVT_REAL SQL_CVT_VARCHAR SQL_CVT_LONGVAR SQL_CVT_BINARY SQL_CVT_BINARY SQL_CVT_BIT SQL_CVT_TINYINT SQL_CVT_TIMESTAN SQL_CVT_LONGVAR SQL_CVT_WCHAR SQL_CVT_WLONGVA
SQL_CORRELATION_NAME	SQL_CN_ANY
SQL_CREATE_ASSERTION	FALSE
SQL_CREATE_CHARACTER_SET	FALSE
SQL_CREATE_COLLATION	FALSE
SQL_CREATE_DOMAIN	FALSE
SQL_CREATE_SCHEMA	SQL_CS_AUTHORIZA SQL_CS_CREATE_SC
SQL_CREATE_TABLE	SQL_CT_CREATE_TA
SQL_CREATE_TRANSLATION	FALSE
SQL_CREATE_VIEW	SQL_CV_CHECK_OP SQL_CV_CREATE_VI
SQL_CURSOR_COMMIT_BEHAVIOR	SQL_CB_CLOSE
SQL_CURSOR_ROLLBACK_BEHAVIOR	SQL_CB_CLOSE
SQL_CURSOR_SENSITIVITY	SQL_SENSITIVE
SQL_DATA_SOURCE_NAME	Current data source nam by <i>StringLengthPtr</i> to 0 specify a data source na
SQL_DATA_SOURCE_READ_ONLY	Depends on setting of co SQL_ATTR_ACCESS_
SQL_DATABASE_NAME	The connection's curren
SQL_DBMS_NAME	"Microsoft SQL Server"
SQL_DBMS_VER	The version number of t

	SQL Server.
SQL_DEFAULT_TXN_ISOLATION	SQL_TXN_READ_CO
SQL_DESCRIBE_PARAMETER	"Y"
SQL_DRIVER_NAME	"Sqlsrv32.dll"
SQL_DRIVER_ODBC_VER	The driver's supported (
SQL_DRIVER_VER	The version number of t
SQL_DROP_ASSERTION	FALSE
SQL_DROP_CHARACTER_SET	FALSE
SQL_DROP_COLLATION	FALSE
SQL_DROP_DOMAIN	FALSE
SQL_DROP_SCHEMA	DROP SCHEMA not su
SQL_DROP_TABLE	SQL_DT_DROP_TABI
SQL_DROP_TRANSLATION	FALSE
SQL_DROP_VIEW	SQL_DV_DROP_VIEV
SQL_DYNAMIC_CURSOR_ATTRIBUTES1	SQL_CA1_ABSOLUT
	SQL_CA1_BULK_AD
	SQL_CA1_LOCK_NO
	SQL_CA1_NEXT
	SQL_CA1_POS_DELE
	SQL_CA1_POS_POSI
	SQL_CA1_POS_REFR
	SQL_CA1_POS_UPDA
	SQL_CA1_POSITIONI
	SQL_CA1_POSITIONI
	SQL_CA1_RELATIVE
	SQL_CA1_SELECT_F
SQL_DYNAMIC_CURSOR_ATTRIBUTES2	SQL_CA2_LOCK_CO
	SQL_CA2_MAX_ROV
	SQL_CA2_OPT_ROW
	SQL_CA2_OPT_VALU
	SQL_CA2_KEAD_ON
	SQL_CA2_READ_ON

	SOL CAR SENSITIVI
	SQL_CA2_SENSITIVI
	SQL_CA2_SENSITIVI
COL EXPRESSIONS IN ODDERRY	SQL_CA2_SIMULATE
SQL_EXPRESSIONS_IN_ORDERBY	"Y"
SQL_FETCH_DIRECTION	SQL_FD_FETCH_ABS
	SQL_FD_FETCH_BO(
	SQL_FD_FETCH_FIR
	SQL_FD_FETCH_LAS
	SQL_FD_FETCH_NE
	SQL_FD_FETCH_PRI
	SQL_FD_FETCH_REL
SQL_FILE_USAGE	SQL_FILE_NOT_SUP
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL_CA1_NEXT
	SQL_CA1_POSITIONI
	SQL_CA1_POSITION
	SQL_CA1_SELECT_F
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL_CA2_LOCK_CO
	SQL_CA2_MAX_ROV
	SQL_CA2_OPT_ROW
	SQL_CA2_OPT_VALU
	SQL_CA2_READ_ON
SQL_GETDATA_EXTENSIONS	SQL_GD_BLOCK
SQL_GROUP_BY	SQL_GB_GROUP_BY
SQL IDENTIFIER CASE	SQL_IC_MIXED if cor
· ~ · · · · · · · · · · · · · ·	running a case-insenstiv
	_
	SQL_IC_SENSITIVE i
	running case-sensitive s
SQL_IDENTIFIER_QUOTE_CHAR	" (the double quote char
SQL_INDEX_KEYWORDS	SQL_IK_ASC
	SQL_IK_DESC
SQL_INFO_SCHEMA_VIEWS	Request not supported b
h	

SQL_INFO_SS_NETLIB_NAME	SQL Server ODBC driv name of the network lib connection.
SQL_INTEGRITY	"Y"
SQL_KEYSET_CURSOR_ATTRIBUTES1	SQL_CA1_ABSOLUTI SQL_CA1_BOOKMAH SQL_CA1_BULK_ADI SQL_CA1_BULK_DEI SQL_CA1_BULK_FET SQL_CA1_BULK_FET SQL_CA1_BULK_UPI SQL_CA1_LOCK_NO SQL_CA1_LOCK_NO SQL_CA1_POS_DELE SQL_CA1_POS_DELE SQL_CA1_POS_REFR SQL_CA1_POS_UPDA SQL_CA1_POSITIONI SQL_CA1_POSITIONI SQL_CA1_POSITIONI SQL_CA1_RELATIVE SQL_CA1_SELECT_F
SQL_KEYSET_CURSOR_ATTRIBUTES2	SQL_CA2_CRC_EXA(SQL_CA2_LOCK_CO] SQL_CA2_MAX_ROV SQL_CA2_MAX_ROV SQL_CA2_MAX_ROV SQL_CA2_MAX_ROV SQL_CA2_MAX_ROV SQL_CA2_OPT_ROW SQL_CA2_OPT_VALU SQL_CA2_OPT_VALU SQL_CA2_READ_ON SQL_CA2_SENSITIVI SQL_CA2_SENSITIVI
SQL_KEYWORDS	BREAK BROWSE BULK CHECKPOINT

CLUSTERED COMMITTED COMPUTE CONFIRM CONTROLROW DATABASE DBCC DISK DISTRIBUTED DUMMY DUMP ERRLVL ERROREXIT EXIT FILE FILLFACTOR FLOPPY HOLDLOCK IDENTITY INSERT IDENTITYCOL IF KILL LINENO LOAD MIRROREXIT NONCLUSTERED OFF OFFSETS ONCE OVER PERCENT PERM PERMANENT PLAN PRINT PROC PROCESSEXIT

	RAISERROR
	READ
	READTEXT
	RECONFIGURE
	REPEATABLE
	RETURN
	ROWCOUNT RULE
	SAVE
	SERIALIZABLE
	SETUSER
	SHUTDOWN
	STATISTICS
	ТАРЕ
	TEMP
	TEXTSIZE
	TRAN
	TRIGGER
	TRUNCATE
	TSEQUEL
	UNCOMMITTED
	UPDATETEXT
	USE
	WAITFOR
	WHILE
	WRITETEXT
SQL_LIKE_ESCAPE_CLAUSE	"Y"
SQL_LOCK_TYPES	SQL_LCK_NO_CHAN
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	1
SQL_MAX_BINARY_LITERAL_LEN	131072
SQL_MAX_CATALOG_NAME_LEN	128/30*
SQL_MAX_CHAR_LITERAL_LEN	131072
SQL_MAX_COLUMN_NAME_LEN	128/30*
SQL_MAX_COLUMNS_IN_GROUP_BY	16
SQL_MAX_COLUMNS_IN_INDEX	16
SQL_MAX_COLUMNS_IN_ORDER_BY	16

SQL_MAX_COLUMNS_IN_SELECT	4000
SQL_MAX_COLUMNS_IN_TABLE	250
SQL_MAX_CONCURRENT_ACTIVITIES	1
SQL_MAX_CURSOR_NAME_LEN	128/30*
SQL_MAX_DRIVER_CONNECTIONS	0
SQL_MAX_IDENTIFIER_LEN	128/30*
SQL_MAX_INDEX_SIZE	127
SQL_MAX_PROCEDURE_NAME_LEN	134/36* (SQL Server pr the name (128 bytes in 5 bytes in earlier versions and a 5 digit number.)
SQL_MAX_ROW_SIZE	8062/1962*
SQL_MAX_ROW_SIZE_INCLUDES_LONG	"N"
SQL_MAX_SCHEMA_NAME_LEN	128/30*
SQL_MAX_STATEMENT_LEN	131072
SQL_MAX_TABLE_NAME_LEN	128/30*
SQL_MAX_TABLES_IN_SELECT	16
SQL_MAX_USER_NAME_LEN	128/30*
SQL_MAX_OWNER_NAME_LEN	128/30*
SQL_MAX_QUALIFIER_NAME_LEN	128/30*
SQL_MULT_RESULT_SETS	"Y"
SQL_MULTIPLE_ACTIVE_TXN	"Y"
SQL_NEED_LONG_DATA_LEN	"Y"
SQL_NON_NULLABLE_COLUMNS	SQL_NNC_NON_NUL
SQL_NULL_COLLATION	SQL_NC_LOW
SQL_NUMERIC_FUNCTIONS	SQL_FN_NUM_ABS
	SQL_FN_NUM_ACOS
	SQL_FN_NUM_ASIN
	SQL_FN_NUM_ATAN
	SQL_FN_NUM_ATAN
	SQL_FN_NUM_CEILI
	SQL_FN_NUM_COS
	SQL_FN_NUM_COT SQL_FN_NUM_DEGR

	SQL_FN_NUM_EXP
	SQL_FN_NUM_FLOO
	SQL_FN_NUM_LOG
	SQL_FN_NUM_LOG1
	SQL_FN_NUM_MOD
	SQL_FN_NUM_PI
	SQL_FN_NUM_POWE
	SQL_FN_NUM_RADL
	SQL_FN_NUM_RANE
	SQL_FN_NUM_ROUN
	SQL_FN_NUM_SIGN
	SQL_FN_NUM_SIN
	SQL_FN_NUM_SQRT
	SQL_FN_NUM_TAN
SQL_ODBC_API_CONFORMANCE	SQL_OAC_LEVEL2
SQL_ODBC_INTERFACE_CONFORMANCE	SQL_OIC_LEVEL2 wh
	instance of SQL Server
	SQL_OIC_CORE when
	version 6.5 or earlier.
SQL_ODBC_SAG_CLI_CONFORMANCE	SQL_OSCC_NOT_CO
SQL_ODBC_SQL_CONFORMANCE	SQL_OSC_CORE
SQL_ODBC_SQL_OPT_IEF	"Y"
SQL_ODBC_VER	Current version number
	Manager.
SQL_OJ_CAPABILITIES	SQL_OJ_ALL_COMPA
	SQL_OJ_FULL
	SQL_OJ_INNER
	SQL_OJ_LEFT
	SQL_OJ_NESTED
	SQL_OJ_NOT_ORDEI
	SQL_OJ_RIGHT
SQL_OUTER_JOINS	"Y"
SQL_ORDER_BY_COLUMNS_IN_SELECT	"N"
SQL_OWNER_USAGE	SQL_OU_DML_STAT
	SQL_OU_INDEX_DEF

	SQL_OU_PRIVILEGE
	SQL_OU_PROCEDUR
	SQL_OU_TABLE_DEI
SQL_PARAM_ARRAY_ROW_COUNTS	SQL_PARC_BATCH
SQL_PARAM_ARRAY_SELECTS	SQL_PAS_BATCH
SQL_POS_OPERATIONS	SQL_POS_ADD
	SQL_POS_DELETE
	SQL_POS_POSITION
	SQL_POS_REFRESH
	SQL_POS_UPDATE
SQL_POSITIONED_STATEMENTS	SQL_PS_POSITIONEI
	SQL_PS_POSITIONEI
	SQL_PS_SELECT_FO
SQL_PROCEDURE_TERM	"stored procedure"
SQL_PROCEDURES	"Y"
SQL_QUALIFIER_USAGE	SQL_CU_DML_STATI
	SQL_CU_PROCEDUR
	SQL_CU_TABLE_DEI
SQL_QUOTED_IDENTIFIER_CASE	SQL_IC_MIXED when
	running a case-insensiti
	SQL_IC_SENSITIVE v
	server running a case-se
SQL_ROW_UPDATES	"N"
-	
SQL_SCHEMA_TERM	"owner"
SQL_SCHEMA_USAGE	SQL_OU_DML_STATI
	SQL_OU_INDEX_DEF
	SQL_OU_PRIVILEGE
	SQL_OU_PROCEDUR
	SQL_OU_TABLE_DEI
SQL_SCROLL_OPTIONS	SQL_SO_DYNAMIC
	SQL_SO_FORWARD_
	SQL_SO_KEYSET_DF
	SQL_SO_STATIC
SQL_SCROLL_CONCURRENCY	SQL_SCCO_LOCK

	SQL_SCCO_OPT_ROV SQL_SCCO_OPT_VAL
COL CEADOU DATTEDNI ECCADE	SQL_SCCO_READ_O
SQL_SEARCH_PATTERN_ESCAPE	"\"
SQL_SERVER_NAME	The connection's server
SQL_SPECIAL_CHARACTERS	Depends on SQL Server
SQL_SQL92_DATETIME_FUNCTIONS	FALSE
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	FALSE
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	FALSE
SQL_SQL92_GRANT	SQL_SG_WITH_GRA1
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	FALSE
SQL_SQL92_PREDICATES	SQL_SP_EXISTS SQL SQL_SP_ISNULL
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL_SRJO_CROSS_J(SQL_SRJO_FULL_OU SQL_SRJO_INNER_JC SQL_SRJO_LEFT_OU SQL_SRJO_RIGHT_O SQL_SRJO_UNION_J(
SQL_SQL92_REVOKE	SQL_SR_GRANT_OP.
SQL_SQL92_ROW_VALUE_CONSTRUCTOR	SQL_SRVC_DEFAULT SQL_SRVC_NULL SQL_SRVC_ROW_SU SQL_SRVC_VALUE_E
SQL_SQL92_STRING_FUNCTIONS	SQL_SSF_LOWER SQL_SSF_UPPER
SQL_SQL92_VALUE_EXPRESSIONS	SQL_SVE_CASE SQL_SVE_CAST SQL_SVE_COALESCI SQL_SVE_NULLIF
SQL_STANDARD_CLI_CONFORMANCE	SQL_SCC_ISO92_CLI
SQL_STATIC_CURSOR_ATTRIBUTES1	SQL_CA1_ABSOLUTI SQL_CA1_BOOKMAF SQL_CA1_BULK_FET SQL_CA1_BULK_NO

	SQL_CA1_NEXT
	SQL_CA1_POS_POSI
	SQL_CA1_POS_REFR
	SQL_CA1_RELATIVE
SQL_STATIC_CURSOR_ATTRIBUTES2	SQL_CA2_CRC_EXA(
	SQL_CA2_MAX_ROV
	SQL_CA2_READ_ON
SQL_STATIC_SENSITIVITY	SQL_SS_ADDITIONS
	SQL_SS_UPDATES
SQL_STRING_FUNCTIONS	SQL_FN_STR_ASCII
·	SQL_FN_STR_BIT_LI
	SQL_FN_STR_CHAR
	SQL_FN_STR_CONC
	SQL_FN_STR_DIFFEI
	SQL_FN_STR_INSER
	SQL_FN_STR_LCASE
	SQL_FN_STR_LEFT
	SQL_FN_STR_LENG1
	SQL_FN_STR_LOCAT
	SQL_FN_STR_LTRIM
	SQL_FN_STR_OCTET
	SQL_FN_STR_REPEA
	SQL_FN_STR_RIGHT
	SQL_FN_STR_RTRIM
	SQL_FN_STR_SOUNI
	SQL_FN_STR_SPACE
	SQL_FN_STR_SUBST
	SQL_FN_STR_UCASE
SQL_SUBQUERIES	SQL_SQ_COMPARISC
	SQL_SQ_CORRELATI
	SQL_SQ_EXISTS
	SQL_SQ_IN
	SQL_SQ_QUANTIFIE

SQL_SYSTEM_FUNCTIONS	SQL FN SYS DBNAI
	SQL_FN_SYS_IFNUL
	SQL_FN_SYS_USERN
SOL TADLE TEDM	"table"
SQL_TABLE_TERM	
SQL_TIMEDATE_ADD_INTERVALS	SQL_FN_TSI_DAY
	SQL_FN_TSI_FRAC_S
	SQL_FN_TSI_HOUR
	SQL_FN_TSI_MINUT
	SQL_FN_TSI_MONTH
	SQL_FN_TSI_QUART
	SQL_FN_TSI_SECON
	SQL_FN_TSI_WEEK
	SQL_FN_TSI_YEAR
SQL_TIMEDATE_DIFF_INTERVALS	SQL_FN_TSI_DAY
	SQL_FN_TSI_FRAC_5
	SQL_FN_TSI_HOUR
	SQL_FN_TSI_MINUT
	SQL_FN_TSI_MONTH
	SQL_FN_TSI_QUART
	SQL_FN_TSI_SECON
	SQL_FN_TSI_WEEK
	SQL_FN_TSI_YEAR
SQL_TIMEDATE_FUNCTIONS	SQL_FN_TD_CURDA
	SQL_FN_TD_CURREI
	SQL_FN_TD_CURREI
	SQL_FN_TD_CURREI
	SQL_FN_TD_CURTIM
	SQL_FN_TD_DAYNA
	SQL_FN_TD_DAYOF1
	SQL_FN_TD_DAYOF
	SQL_FN_TD_DAYOF
	SQL_FN_TD_EXTRA(
	SQL_FN_TD_HOUR
	SQL_FN_TD_MINUTI
	SQL_FN_TD_MONTH
	SQL_FN_TD_MONTH

	SQL_FN_TD_NOW SQL_FN_TD_QUARTI SQL_FN_TD_SECONI SQL_FN_TD_TIMEST SQL_FN_TD_TIMEST SQL_FN_TD_WEEK
	SQL_FN_TD_YEAR
SQL_TXN_CAPABLE	SQL_TC_ALL
SQL_TXN_ISOLATION_OPTION	SQL_TXN_READ_CO
	SQL_TXN_READ_UN
	SQL_TXN_REPEATAI
	SQL_TXN_SERIALIZ
SQL_UNION	SQL_U_UNION
	SQL_U_UNION_ALL
SQL_USER_NAME	The current username.
* Dependent on SQL Server version. First value v	when connected to SQL Serve
value for all earlier versions.	

SQLGetStmtAttr

The Microsoft® SQL Server[™] ODBC driver extends **SQLGetStmtAttr** to expose driver-specific statement attributes. All driver-specific attributes are SQLINTEGER values.

The SQL Server ODBC driver SQL_TEXTPTR_LOGGING attribute exposes logging of operations on columns containing **text** or **image** data.

Value	Description
_	Logging operations performed on text , ntext , and image data is disabled.
SQL_TL_ON	Default. Logging of operations performed on text , ntext , and image data is enabled.

The SQL_SOPT_SS_CURRENT_COMMAND attribute exposes the current command of a command batch. The return is an integer specifying the location of the command in the batch.

SQL_SOPT_SS_HIDDEN_COLUMNS exposes, in the result set, columns hidden in a SQL Server SELECT FOR BROWSE statement. The driver does not expose these columns by default.

Value	Description
SQL_HC_OFF	Default. FOR BROWSE columns are hidden from the
	result set.
SQL_HC_ON	Exposes FOR BROWSE columns.

SQL_SOPT_SS_NCOUNT_STATUS indicates the current setting of the NOCOUNT option, which controls whether SQL Server reports the numbers of rows affected by a statement when **SQLRowCount** is called.

Value	Description
SQL_NC_OFF	NOCOUNT is OFF. SQLRowCount returns number of
	rows affected.
SQL_NC_ON	NOCOUNT is ON. The counts of rows affected is not

returned by SQLRowCount .

SQLGetTypeInfo

The Microsoft® SQL Server[™] ODBC driver reports the additional column USERTYPE in the result set of **SQLGetTypeInfo**. USERTYPE reports the DB-Library data type definition and is useful to developers porting existing DB-Library applications to ODBC.

SQL Server treats identity as an attribute, whereas ODBC treats it as a data type. To resolve this mismatch, **SQLGetTypeInfo** returns the data types: **int identity**, **smallint identity**, **tinyint identity**, **decimal identity**, and **numeric identity**. The **SQLGetTypeInfo** result set column AUTO_UNIQUE_VALUE reports the value TRUE for these data types.

SQLMoreResults

SQLMoreResults allows the application to retrieve multiple sets of result rows. A Transact-SQL SELECT statement containing a COMPUTE clause, or a submitted batch of ODBC or Transact-SQL statements, causes the Microsoft® SQL Server[™] ODBC driver to generate multiple result sets. SQL Server does not allow creation of a server cursor to process the results in either case. Therefore, the developer must ensure that the ODBC statement is blocking. The developer must exhaust the returned data or cancel the ODBC statement before he or she can process data from other active statements on the connection.

The developer can determine properties of the result sets columns and rows that are generated by the COMPUTE clause of a SQL Server SELECT statement. For more detail, see **SQLColAttribute**.

When **SQLMoreResults** is called with unfetched data rows in the result set, those rows are lost, and row data from the next result row set is made available.

Examples

```
void GetComputedRows
 (
 SQLHSTMT hStmt
 )
  {
 SQLUSMALLINT nCols;
 SQLUSMALLINT nCol;
 PODBCSETINFO
                 pODBCSetInfo = NULL;
 SQLRETURN
                sRet;
 UINT
            nRow;
 SQLINTEGER
                nComputes = 0;
 SQLINTEGER
                nSet;
 BYTE*
             pValue;
```

// If SQLNumResultCols failed, then some error occurred in

```
// statement execution. Exit.
if (!SQL_SUCCEEDED(SQLNumResultCols(hStmt, (SQLSMALL
        {
        goto EXIT;
        }
```

// Determine the presence of COMPUTE clause result sets. The SQL// Server ODBC driver uses column attributes to report multiple

- // sets. The column number must be less than or equal to the
- // number of columns returned. You are guaranteed to have at least
- // one, so use '1' for the SQLColAttribute ColumnNumber
 // parameter.

```
SQLColAttribute(hStmt, 1, SQL_CA_SS_NUM_COMPUTES, NULL, 0, NULL, (SQLPOINTER) &nComputes);
```

// Create a result info structure pointer array, one element for

```
// the normal result rows and one for each compute result set.
```

```
// Initialize the array to NULL pointers.
```

```
pODBCSetInfo = new ODBCSETINFO[1 + nComputes];
```

```
if (GetColumnsInfo(hStmt, pODBCSetInfo[nSet].nCols,
   &(pODBCSetInfo[nSet].pODBCColInfo)) == SQL_ERROR
   {
    goto EXIT;
   }
}
```

// Get memory for bound return values if required.
if (pODBCSetInfo[nSet].pRowValues == NULL)
 {
 CreateBindBuffer(&(pODBCSetInfo[nSet]));
 }

// Rebind columns each time the result set changes. myBindCols(hStmt, pODBCSetInfo[nSet].nCols, pODBCSetInfo[nSet].pODBCColInfo, pODBCSetInfo[nSet].pRowValues);

// Set for ODBC row array retrieval. Fast retrieve for all // sets. COMPUTE row sets have only a single row, but // normal rows can be retrieved in blocks for speed. SQLSetStmtAttr(hStmt, SQL_ATTR_ROW_BIND_TYPE, (void*) pODBCSetInfo[nSet].nResultWidth, SQL_IS_UINTEC SQLSetStmtAttr(hStmt, SQL_ATTR_ROW_ARRAY_SIZE, (void*) pODBCSetInfo[nSet].nRows, SQL_IS_UINTEGER); SQLSetStmtAttr(hStmt, SQL_ATTR_ROWS_FETCHED_PTR, (void*) &nRowsFetched, sizeof(SQLINTEGER));

while (TRUE)

{

// In ODBC 3.x, SQLFetch supports arrays of bound rows or

// columns. SQLFetchScroll (or ODBC 2.x SQLExtendedFetch

// is not necessary to support fastest retrieval of

```
// data rows.
  if (!SQL_SUCCEEDED(sRet = SQLFetch(hStmt)))
    break;
     }
  for (nRow = 0; nRow < (UINT) nRowsFetched; nRow++)
     ł
    for (nCol = 0; nCol < pODBCSetInfo[nSet].nCols;</pre>
         nCol++)
       {
      // Processing row and column values...
       }
     }
  }
// sRet is not SQL_SUCCESS and is not SQL_SUCCESS_WITH
// If it's SQL NO DATA, then continue. If it's an
// error state, stop.
if (sRet != SQL_NO_DATA)
  ł
  break;
  }
// If there's another set waiting, determine the result set
// indicator. The indicator is 0 for regular row sets or an
// ordinal indicating the COMPUTE clause responsible for the
// set.
if (SQLMoreResults(hStmt) == SQL_SUCCESS)
  sRet = SQLColAttribute(hStmt, 1, SQL_CA_SS_COMPUTE_)
    NULL, 0, NULL, (SQLPOINTER) &nSet);
  }
```

```
else
{
break;
}
}
```

EXIT:

// Clean-up anything dynamically allocated and return.
return;
}

See Also

<u>SELECT</u>

SQLColAttribute

SQLNativeSql

The Microsoft® SQL Server[™] ODBC driver satisfies **SQLNativeSql** requests without visiting the server. The function will efficiently test the syntax of SQL statements. Syntax checking does not determine if identifiers or the results of expressions in the SQL are valid, and SQL Server native SQL returned by **SQLNativeSql** can fail to run.

SQLNumResultCols

For executed statements, the Microsoft® SQL Server[™] ODBC driver does not visit the server to report the number of columns in a result set. In this case, **SQLNumResultCols** does not cause a server roundtrip. Like **SQLDescribeCol** and **SQLColAttribute**, calling **SQLNumResultCols** on prepared but not executed statements generates a server roundtrip.

When a Transact-SQL statement or statement batch returns multiple result row sets, it is possible for the number of result set columns to change from one set to another. **SQLNumResultCols** should be called for each set. When the number of columns changes, the application should rebind data values prior to fetching row results. For more information about handling multiple result set returns, see **SQLMoreResults**.

See Also

SQLMoreResults

SQLPrepare

Note Microsoft[®] SQL Server[™] 2000 supports the prepare/execute model of ODBC. The following discussion of **SQLPrepare/SQLExecute** behavior is applicable only to versions of SQL Server earlier than 7.0.

The SQL Server ODBC driver creates a temporary stored procedure from prepared SQL statements. Stored procedures are an efficient way to execute a statement multiple times, but stored procedure creation is more expensive than simple statement execution. As a general rule, consider using **SQLPrepare** and **SQLExecute** if the application will submit an SQL statement more than three times.

A temporary stored procedure created by **SQLPrepare** is named **#odbc#***useridentifier*, where *useridentifier* is up to 6 characters of the user-name concatenated with up to 8 digits that identify the procedure.

SQLPrepare creates the temporary stored procedure if all parameter values have been bound or if the SQL statement does not contain parameters. **SQLExecute** creates the procedure if all parameters were not bound when **SQLPrepare** was called.

SQLPrepare can create stored procedures more efficiently than **SQLExecute**, and it is suggested that **SQLBindParameter** be used to bind parameter variables prior to calling **SQLPrepare**.

If the CREATE PROCEDURE statement used to generate a temporary stored procedure returns an error, **SQLPrepare** or **SQLExecute** submits the statement to SQL Server with the SET NOEXEC or SET PARSEONLY option enabled (depending on the statement type). SQL Server checks the syntax of the statement and returns any errors.

SQLExecute can return any ODBC SQLSTATE and any SQL Server error that can be returned by **SQLPrepare**.

The SQL Server ODBC driver creates a new temporary stored procedure if the *InputOutputType*, *ParameterType*, *ColumnSize*, or *DecimalDigits* values are altered in calls to **SQLBindParameter** on a prepared statement. A new temporary stored procedure will not be created when bound parameters are

pointed to new buffers in client memory, the length of client memory is changed, or the pointer to the length or indicator value for the parameter is altered.

If a connection cannot create a stored procedure for any reason (such as lack of permission), the SQL Server ODBC driver does not use a stored procedure but, instead, submits the SQL statement each time **SQLExecute** is called.

By default, the SQL Server ODBC driver drops temporary stored procedures when the connection is broken (**SQLDisconnect** is called for the connection). This may present problems if the connection is expected to remain open indefinitely. The default behavior can be changed using the driver-specific connection option SQL_USE_PROCEDURE_FOR_PREPARE.

Note If SET NOCOUNT ON has been executed, multiple statements embedded in a stored procedure do not create multiple result sets as they should. Row counts generated by SQL statements inside a stored procedure are ignored by the driver.

See Also

SQLBindParameter
SQLSetConnectAttr

SQLPrimaryKeys

SQLPrimaryKeys uses the catalog stored procedure **sp_pkeys** to report primary key participants from a table. Though a table may have a column or columns that can serve as unique row identifiers, tables created without a PRIMARY KEY constraint return an empty result set to **SQLPrimaryKeys**. The ODBC function **SQLSpecialColumns** reports row identifier candidates for tables without primary keys.

The following table shows **SQLPrimaryKeys** parameter mapping for **sp_pkeys** stored procedure execution.

SQLPrimaryKeys parameter name	sp_pkeys parameter name
CatalogName	table_qualifier
SchemaName	table_owner
TableName	table_name

SQLPrimaryKeys returns SQL_SUCCESS whether or not values exist for *CatalogName, SchemaName*, or *TableName* parameters. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLPrimaryKeys can be executed on a static server cursor. An attempt to execute **SQLPrimaryKeys** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

The Microsoft® SQL Server[™] ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *CatalogName* parameter: *Linked_Server_Name.Catalog_Name*

See Also

<u>sp_pkeys</u>

SQLSpecialColumns

SQLProcedureColumns

SQLProcedureColumns uses the catalog stored procedure **sp_sproc_columns** to report the attributes of stored procedure columns.

The following table shows **SQLProcedureColumns** parameter mapping for **sp_sproc_columns** stored procedure execution.

SQLProcedureColumns parameter	sp_sproc_columns parameter
name	name
CatalogName	procedure_qualifier
SchemaName	procedure_owner
ProcName	procedure_name
ColumnName	column_name

SQLProcedureColumns returns one row reporting the return value attributes of all Microsoft® SQL Server[™] stored procedures.

SQLProcedureColumns returns SQL_SUCCESS whether or not values exist for *CatalogName, SchemaName, ProcName*, or *ColumnName* parameters. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLProcedureColumns can be executed on a static server cursor. An attempt to execute **SQLProcedureColumns** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

See Also

sp_sproc_columns

SQLProcedures

SQLProcedures uses the catalog stored procedure **sp_stored_procedures** to report the names of stored procedures in a Microsoft® SQL Server[™] database.

The following table shows **SQLProcedures** parameter mapping for **sp_stored_procedures** stored procedure execution.

SQLProcedures parameter	sp_stored_procedures parameter
name	name
CatalogName	procedure_qualifier
SchemaName	procedure_owner
ProcName	procedure_name

All SQL Server stored procedures return a value. **SQLProcedures** reports SQL_PT_FUNCTION for the result set column PROCEDURE_TYPE.

SQLProcedures returns SQL_SUCCESS whether or not values exist for *CatalogName, SchemaName*, or *ProcName* parameters. **SQLFetch** returns SQL_NO_DATA when invalid values are used in these parameters.

SQLProcedures can be executed on a static server cursor. An attempt to execute **SQLProcedures** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO, indicating that the cursor type has been changed.

SQLProcedures returns information about any tables whose names match *ProcName* and are owned by the current user.

See Also

sp_stored_procedures

SQLPutData

Using **SQLPutData** to send more than 65,535 bytes of data (for Microsoft® SQL Server[™] version 4.21a) or 400 KB of data (for SQL Server version 6.0 and later) for a SQL_LONGVARCHAR (**text**), SQL_WLONGVARCHAR (**ntext**) or SQL_LONGVARBINARY (**image**) column, imposes the following restrictions:

- The referenced parameter can be an *insert_value* in an INSERT statement.
- The referenced parameter can be an *expression* in the SET clause of an UPDATE statement.

Canceling a sequence of **SQLPutData** calls that provide data to a server running SQL Server in blocks causes a partial update of the column's value when using version 6.5 or earlier. The **text**, **ntext**, or **image** column referenced when **SQLCancel** was called will be set to an intermediate "place holder" value.

SQLRowCount

When arrays of parameter values are bound for statement execution, **SQLRowCount** returns SQL_ERROR if any row of parameter values generates an error condition in statement execution. No value is returned through the *RowCountPtr* argument of the function.

The application can take advantage of the SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute to capture the number of parameters processed prior to the error occurring.

Further, the application can use an array of status values, bound by using the SQL_ATTR_PARAM_STATUS_PTR statement attribute, to capture the array offsets of offending parameter rows. The application can traverse the status array to determine the actual number of rows processed.

SQLSetConnectAttr

The Microsoft® SQL Server[™] ODBC driver ignores the setting of SQL_ATTR_CONNECTION_TIMEOUT. The SQL Server ODBC driver will not time out on any operations other than login and query processing.

The SQL Server ODBC driver implements repeatable read transaction isolation as serializable. Setting SQL_ATTR_TXN_ISOLATION to SQL_TXN_REPEATABLE_READ is exactly equivalent to setting the transaction isolation attribute to SQL_TXN_SERIALIZABLE.

Promoting ODBC statement attributes to connection attributes can have unintended consequences. Statement attributes that request server cursors for result set processing can be promoted to the connection. For example, setting the ODBC statement attribute SQL_ATTR_CONCURRENCY to a value more restrictive than the default SQL_CONCUR_READ_ONLY directs the driver to use dynamic cursors for all statements submitted on the connection. Executing an ODBC catalog function on a statement on the connection returns SQL_SUCCESS_WITH_INFO and a diagnostic record indicating that the cursor behavior has been changed to read-only. Attempting to execute a Transact-SQL SELECT statement containing a COMPUTE clause on the same connection fails.

The SQL Server ODBC driver supports a number of driver-specific extensions to ODBC connection attributes defined in Odbcss.h. The SQL Server ODBC driver may require that the attribute be set prior to connection, or it may ignore the attribute if it is already set. The following table lists restrictions.

	Set before or after
SQL Server attribute	connection to server
SQL_COPT_SS_ANSI_NPW	Before
SQL_COPT_SS_ATTACHDBFILENAME	Before
SQL_COPT_SS_BCP	Before
SQL_COPT_SS_BROWSE_CONNECT	Before
SQL_COPT_SS_BROWSE_SERVER	Before
SQL_COPT_SS_CONCAT_NULL	Before

SQL_COPT_SS_ENLIST_IN_DTC	After
SQL_COPT_SS_ENLIST_IN_XA	After
SQL_COPT_SS_FALLBACK_CONNECT	Before
SQL_COPT_SS_INTEGRATED_SECURITY	Before
SQL_COPT_SS_PERF_DATA	After
SQL_COPT_SS_PERF_DATA_LOG	After
SQL_COPT_SS_PERF_DATA_LOG_NOW	After
SQL_COPT_SS_PERF_QUERY	After
SQL_COPT_SS_PERF_QUERY_INTERVAL	After
SQL_COPT_SS_PERF_QUERY_LOG	After
SQL_COPT_SS_PRESERVE_CURSORS	Before
SQL_COPT_SS_QUOTED_IDENT	Either
SQL_COPT_SS_TRANSLATE	Either
SQL_COPT_SS_USE_PROC_FOR_PREP	Either
SQL_COPT_SS_USER_DATA	Either
SQL_COPT_SS_WARN_ON_CP_ERROR	Before

SQL_COPT_SS_ANSI_NPW

SQL_COPT_SS_ANSI_NPW enables or disables the use of SQL-92 handling of NULL in comparisons and concatenation, character data type padding, and warnings. For more information, see SET ANSI_NULLS, SET ANSI_PADDING, SET ANSI_WARNINGS, and SET CONCAT_NULL_YIELDS_NULL.

Value	Description
SQL_AD_ON	Default. The connection uses SQL-92 default behavior
	handling NULL, padding, and warnings.
SQL_AD_OFF	The connection uses SQL Server-defined handling of
	NULL, character data type padding, and warnings.

SQL_COPT_SS_ATTACHDBFILENAME

SQL_COPT_SS_ATTACHDBFILENAME specifies the name of the primary

file of an attachable database. This database is attached and becomes the default database for the connection. To use SQL_COPT_SS_ATTACHDBFILENAME you must specify the name of the database as the value of the connection attribute SQL_ATTR_CURRENT_CATALOG or in the DATABASE = parameter of a **SQLDriverConnect**. If the database was previously attached, SQL Server will not reattach it. This option is not valid when connected to an instance of SQL Server version 6.5 or earlier.

Value	Description
SQLPOINTER to	The string contains the name of the primary file for the
a character string	database to attach. Include the full path name of the file.

SQL_COPT_SS_BCP

SQL_COPT_SS_BCP enables bulk copy functions on a connection. For more information, see <u>Bulk-Copy Functions</u>.

Value	Description
SQL_BCP_OFF	Default. Bulk copy functions are not available on the
	connection.
SQL_BCP_ON	Bulk copy functions are available on the connection.

SQL_COPT_SS_BROWSE_CONNECT

This attribute is used to customize the result set returned by **SQLBrowseConnect**. SQL_COPT_SS_BROWSE_CONNECT enables or disables the return of additional information from an enumerated instance of SQL Server 2000. This can include information such as whether the server is a cluster, names of different instances, and the version number.

Value	Description
SQL_MORE_INFO_NO	Default. In SQL Server version 6.5 and later,
	SQL BrowseConnect returns a list of servers.

SQL_MORE_INFO_YES	In SQL Server versions 6.5 and 7.0,
	SQLBrowseConnect returns a list of servers. In
	SQL Server 2000, SQLBrowseConnect returns
	an extended string of server properties.

SQL_COPT_SS_BROWSE_SERVER

This attribute is used to customize the result set returned by **SQLBrowseConnect**. SQL_COPT_SS_BROWSE_SERVER specifies the server name for which **SQLBrowseConnect** returns the information.

Value	Description
computername	SQLBrowseConnect returns a list of SQL servers on
	the specified computer. Double backslashes (\\) should
	not be used for the server name (for example, instead of
	\\MyServer, MyServer should be used).
NULL	Default. SQLBrowseConnect returns information for
	all servers in the domain.

SQL_COPT_SS_CONCAT_NULL

SQL_COPT_SS_CONCAT_NULL enables or disables the use of SQL-92 handling of NULL when concatenating strings. For more information, see SET CONCAT_NULL_YIELDS_NULL.

Value	Description
SQL_CN_ON	Default. The connection uses SQL-92 default
	behavior for handling NULL values when
	concatenating strings.
SQL_CN_OFF	The connection uses SQL Server-defined behavior for
	handling NULL values when concatenating strings.

SQL_COPT_SS_ENLIST_IN_DTC

The client calls the Microsoft Distributed Transaction Coordinator (MS DTC) OLE **ITransactionDispenser::BeginTransaction** method to begin an MS DTC transaction and create an MS DTC transaction object that represents the transaction. The application then calls **SQLSetConnectAttr** with the SQL_COPT_SS_ENLIST_IN_DTC option to associate the transaction object with the ODBC connection. All related database activity will be performed under the protection of the MS DTC transaction. The application calls **SQLSetConnectAttr** with SQL_DTC_DONE to end the connection's DTC association. For more information, see the MS DTC documentation.

Value	Description
DTC object*	The MS DTC OLE transaction object that specifies the
	transaction to export to SQL Server.
SQL_DTC_DONE	Delimits the end of a DTC transaction.

SQL_COPT_SS_ENLIST_IN_XA

To begin an XA transaction with an XA-compliant Transaction Processor (TP), the client calls the X/Open **tx_begin** function. The application then calls **SQLSetConnectAttr** with a SQL_COPT_SS_ENLIST_IN_XA parameter of TRUE to associate the XA transaction with the ODBC connection. All related database activity will be performed under the protection of the XA transaction. To end an XA association with an ODBC connection, the client must call **SQLSetConnectAttr** with a SQL_COPT_SS_ENLIST_IN_XA parameter of FALSE. For more information, see the Microsoft Distributed Transaction Coordinator documentation.

SQL_COPT_SS_FALLBACK_CONNECT

This attribute is valid only when connected to SQL Server 6.5. It applies only to standby servers. It does not apply to a virtual server in a cluster/failover configuration. SQL_COPT_SS_FALLBACK_CONNECT enables fallback attempts on a connection. When successfully connected to the primary server, the SQL Server ODBC driver automatically determines the current fallback

server and verifies that fallback information is stored in the Registry. If an attempt to connect to a primary server fails (the connection time-out must be greater than 0 for this to occur), the SQL Server ODBC driver will attempt to connect to the fallback server.

Value	Description
SQL_FB_OFF	Default. Fallback connection processing is not performed on connect.
SQL_FB_ON	Fallback connection will be attempted on login time- out.

SQL_COPT_SS_INTEGRATED_SECURITY

SQL_COPT_SS_INTEGRATED_SECURITY forces use of Windows Authentication for access validation on server login. When Windows Authentication is used, the driver ignores user identifier and password values provided as part of **SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect** processing.

Value	Description
-	Default. SQL Server Authentication is used to validate user identifier and password on login.
	Windows Authentication Mode is used to validate a user's access rights to the SQL Server.

SQL_COPT_SS_PERF_DATA

SQL_COPT_SS_PERF_DATA starts or stops performance data logging. The data log file name must be set prior to starting data logging. See SQL_COPT_SS_PERF_DATA_LOG below.

Value	Description
SQL_PERF_START	Starts the driver sampling performance data.
SQL_PERF_STOP	Stops the counters from sampling performance data.

SQL_COPT_SS_PERF_DATA_LOG

SQL_COPT_SS_PERF_DATA_LOG assigns the name of the log file used to record performance data. The log file name is an ANSI or Unicode, null-terminated string depending upon application compilation. The *StringLength* argument should be SQL_NTS.

SQL_COPT_SS_PERF_DATA_LOG_NOW

SQL_COPT_SS_PERF_DATA_LOG_NOW instructs the driver to write a statistics log entry to disk.

SQL_COPT_SS_PERF_QUERY

SQL_COPT_SS_PERF_QUERY starts or stops logging for long running queries. The query log file name must be supplied prior to starting logging. The application can define "long running" by setting the interval for logging.

Value	Description
SQL_PERF_START	Starts long running query logging.
SQL_PERF_STOP	Stops logging of long running queries.

SQL_COPT_SS_PERF_QUERY_INTERVAL

SQL_COPT_SS_PERF_QUERY_INTERVAL sets the query logging threshold in milliseconds. Queries that do not resolve within the threshold are recorded in the long running query log file. There is no upper limit on the query threshold. A query threshold value of zero causes logging of all queries.

SQL_COPT_SS_PERF_QUERY_LOG

SQL_COPT_SS_PERF_QUERY_LOG assigns the name of a log file for recording long running query data. The log file name is an ANSI or Unicode, null-terminated string depending upon application compilation. The *StringLength* argument should be SQL_NTS.

SQL_COPT_SS_PRESERVE_CURSORS

SQL_COPT_SS_PRESERVE_CURSORS defines the behavior of cursors when manual-commit mode is used. The behavior is exposed as transactions are either committed or rolled back using **SQLEndTran**.

Value	Description
SQL_PC_OFF	Default. Cursors are closed on SQLEndTran .
SQL_PC_ON	Cursors remain open on SQLEndTran .

SQL_COPT_SS_QUOTED_IDENT

SQL_COPT_SS_QUOTED_IDENT allows quoted identifiers in ODBC and Transact-SQL statements submitted on the connection. By supplying quoted identifiers, the SQL Server ODBC driver allows otherwise invalid object names such as "My Table," which contains a space character in the identifier. For more information, see SET QUOTED_IDENTIFIER.

Value	Description
SQL_QI_OFF	The SQL Server connection does not allow quoted identifiers in submitted Transact-SQL.
SQL_QI_ON	Default. The connection allows quoted identifiers in submitted Transact-SQL.

SQL_COPT_SS_TRANSLATE

SQL_COPT_SS_TRANSLATE causes the driver to translate characters between the client and server code pages as MBCS data is exchanged. The attribute affects only data stored in SQL Server **char**, **varchar**, and **text** columns.

Value	Description
SQL_XL_OFF	The driver does not translate characters from one code page to another in character data exchanged between the client and the server.

SQL_XL_ON	Default. The driver translates characters from one code
	page to another in character data exchanged between
	the client and the server. The driver automatically
	configures the character translation, determining the
	code page installed on the server and that in use by the
	client.

SQL_COPT_SS_USE_PROC_FOR_PREP

This attribute is only valid when connected to an instance of SQL Server 6.5 or earlier. SQL_COPT_SS_USE_PROC_FOR_PREP defines the use of temporary stored procedures when ODBC and Transact-SQL statements are prepared for execution. For more information about prepared statement execution, see **SQLPrepare**.

Value	Description
SQL_UP_OFF	The driver does not generate stored procedures when
	the application prepares statements.
SQL_UP_ON	Default. The driver generates a temporary stored
	procedure when a statement is prepared. The stored
	procedure is dropped when the application
	disconnects from the server.
SQL_UP_ON_DROP	The driver generates a temporary stored procedure
	when a statement is prepared. The stored procedure
	is dropped when the statement handle is freed.

SQL_COPT_SS_USER_DATA

SQL_COPT_SS_USER_DATA sets the user data pointer. User data is clientowned memory recorded per connection.

SQL_COPT_SS_WARN_ON_CP_ERROR

When this attribute is set to SQL_COPT_YES, you get a warning if there is a

loss of data during a code page conversion. This applies to only data coming from the server.

Example

This example logs performance data.

```
SQLPERF*
           pSQLPERF;
SQLINTEGER nValue;
// See if you are already logging. SQLPERF* will be NULL if not.
SQLGetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA, &pSQLPI
  sizeof(SQLPERF*), &nValue);
if (pSQLPERF == NULL)
  {
  // Set the performance log file name.
  SQLSetConnectAttr(hDbc, SQL COPT SS PERF DATA LOG,
    (SQLPOINTER) "\\My LogDirectory\\MyServerLog.txt", SQL_N
  // Start logging...
  SQLSetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA,
    (SQLPOINTER) SQL_PERF_START, SQL_IS_INTEGER);
  }
else
  ł
  // Take a snapshot now so that your performance statistics are discer.
  SQLSetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA_LOG_N
  }
```

// ...perform some action...

// ...take a performance data snapshot...
SQLSetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA_LOG_NO'

// ...perform more actions...

// ...take another snapshot...
SQLSetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA_LOG_NO'

// ...and disable logging.
SQLSetConnectAttr(hDbc, SQL_COPT_SS_PERF_DATA,
(SQLPOINTER) SQL_PERF_STOP, SQL_IS_INTEGER);

// Continue on...

See Also

Bulk-Copy Functions

SET ANSI_NULLS

SET ANSI_PADDING

SET ANSI_WARNINGS

SET CONCAT_NULL_YIELDS_NULL

SET QUOTED_IDENTIFIER

SQLPrepare

SQLSetEnvAttr

The *ODBC Programmer's Reference* for ODBC 3 defines how ODBC 3.*x* drivers should interpret the **SQLSetEnvAttr** attribute specifications from applications written to either the ODBC 2.*x* or ODBC 3.*x* API. The Microsoft® SQL ServerTM ODBC driver complies with those rules.

One of the attributes controlled by **SQLSetEnvAttr** is whether connection pooling is to be used. If connection pooling is used with the SQL Server ODBC driver, the *DriverCompletion* parameter must be set to SQL_DRIVER_NOPROMPT when connecting with either **SQLDriverConnect** or **SQLConnect**.

SQLSetStmtAttr

The Microsoft® SQL Server[™] ODBC driver does not support the mixed (keyset/dynamic) cursor model. Attempts to set the keyset size using SQL_ATTR_KEYSET_SIZE fail if the value set is not equal to 0.

The application sets SQL_ATTR_ROW_ARRAY_SIZE on all statements to declare the number of rows returned on a **SQLFetch** or **SQLFetchScroll** function call. On statements indicating a server cursor, the driver uses SQL_ATTR_ROW_ARRAY_SIZE to determine the size of the block of rows the server generates to satisfy a fetch request from the cursor. Within the block size of a dynamic cursor, row membership and ordering are fixed if the transaction isolation level is sufficient to ensure repeatable reads of committed transactions. The cursor is completely dynamic outside of the block indicated by this value. Server cursor block size is completely dynamic and can be changed at any point in fetch processing.

The SQL Server ODBC driver also supports the following driver-specific statement attributes.

SQL_SOPT_SS_CURSOR_OPTIONS

Specifies whether the driver will use driver-specific performance options on cursors. **SQLGetData** is not allowed when these options are set. The default setting is SQL_CO_OFF. These options are valid only when connected to an instance of SQL Server version 7.0.

ValuePtr value	Description
SQL_CO_OFF	Default. Disables fast forward-only, read-only cursors
	and autofetch, enables SQLGetData on forward-only,
	read-only cursors. When
	SQL_SOPT_SS_CURSOR_OPTIONS is set to
	SQL_CO_OFF, the cursor type will not change. That
	is, fast forward-only cursor will remain a fast forward-
	only cursor. To change the cursor type, the application
	must now set a different cursor type using
	SQLSetStmtAttr/SQL_ATTR_CURSOR_TYPE.

SQL_CO_FFO	Enables fast forward-only, read-only cursors, disables SQLGetData on forward-only, read-only cursors.
SQL_CO_AF	Enables the autofetch option on any cursor type. When this option is set for a statement handle, SQLExecute or SQLExecDirect generate an implicit SQLFetchScroll (SQL_FIRST). The cursor is opened and the first batch of rows is returned in a single roundtrip to the server.
•	Enables fast forward-only cursors with the autofetch option. It is the same as if both SQL_CO_AF and SQL_CO_FFO are specified.

When these options are set, the server closes the cursor automatically when it detects that the last row has been fetched. The application must still call **SQLFreeStmt**(SQL_CLOSE) or **SQLCloseCursor**, but the driver does not have to send the close notification to the server.

If the select list contains a **text**, **ntext**, or **image** column, the fast forward-only cursor is converted to a dynamic cursor and **SQLGetData** is allowed.

SQL_SOPT_SS_DEFER_PREPARE

This attribute determines whether the statement is prepared immediately or deferred until **SQLExecute**, **SQLDescribeCol** or **SQLDescribeParam** is executed. In SQL Server version 7.0 and earlier, this property is ignored (no deferred prepare).

ValuePtr value	Description
SQL_DP_ON	Default. After calling SQLPrepare , the statement
	preparation is deferred until SQLExecute is called or
	metaproperty operation (SQLDescribeCol or
	SQLDescribeParam) is executed.
SQL_DP_OFF	The statement is prepared as soon as SQLPrepare is
	executed.

SQL_SOPT_SS_REGIONALIZE

The driver uses this attribute to determine data conversion at the statement level. The attribute causes the driver to respect the client locale setting when converting date, time, and currency values to character strings. The conversion is from SQL Server native data types to character strings only.

ValuePtr value	Description
SQL_RE_OFF	Default. The driver does not convert date, time, and currency data to character string data using the client locale setting.
SQL_RE_ON	The driver uses the client locale setting when converting date, time, and currency data to character string data.

Regional conversion settings apply to currency, numeric, date, and time data types. The conversion setting is only applicable to:

- Output conversions when currency, numeric, date, or time values are converted to character strings.
- Bulk copy in operations containing character columns when BCP6xFILEFMT is also set on.

Note When the statement option SQL_SOPT_SS_REGIONALIZE is on, the driver uses the locale registry settings for the current user. The driver does not honor the current thread's locale if the application sets it by, for example, calling SetThreadLocale.

Altering the regional behavior of a data source can cause application failure. An application that parses date strings and expects date strings to appear as defined by ODBC, could be adversely affected by altering this value.

SQL_SOPT_SS_TEXTPTR_LOGGING

Attribute toggles logging of operations on columns containing **text** or **image** data. The default behavior is to log these operations (SQL_TL_ON).

ValuePtr value	Description

-	Disables logging of operations performed on text and image data.	
-	Default. Enables logging of operations performed on text and image data.	

SQL_SOPT_SS_HIDDEN_COLUMNS

Exposes, in the result set, columns hidden in a SQL Server SELECT FOR BROWSE statement. The driver does not expose these columns by default.

ValuePtr value	Description	
SQL_HC_OFF	Default. FOR BROWSE columns are hidden from the	
	result set.	
SQL_HC_ON	Exposes FOR BROWSE columns.	

SQLSpecialColumns

The Microsoft® SQL Server[™] ODBC driver uses the catalog stored procedure **sp_special_columns** to generate the result set for **SQLSpecialColumns**.

When requesting row identifiers (*IdentifierType* SQL_BEST_ROWID), **SQLSpecialColumns** returns an empty result set (no data rows) for any requested scope other than SQL_SCOPE_CURROW. The generated result set indicates that the columns are only valid within this scope.

SQL Server does not support pseudo columns for identifiers. The **SQLSpecialColumns** result set will identify all columns as SQL_PC_NOT_PSEUDO.

SQLSpecialColumns can be executed on a static cursor. An attempt to execute **SQLSpecialColumns** on an updatable (keyset-driven or dynamic) returns SQL_SUCCESS_WITH_INFO indicating the cursor type has been changed.

SQLStatistics

The Microsoft® SQL Server[™] ODBC driver uses the catalog stored procedure **sp_statistics** to provide results for **SQLStatistics**.

SQLStatistics can be executed on a static cursor. An attempt to execute **SQLStatistics** on an updatable (keyset-driven or dynamic) returns SQL_SUCCESS_WITH_INFO indicating the cursor type is changed.

See Also

sp_statistics

SQLTablePrivileges

The Microsoft® SQL Server[™] ODBC driver uses the **sp_table_privileges** catalog stored procedure to satisfy table privilege data requests using **SQLTablePrivileges**.

SQLTablePrivileges can be executed on a static cursor. An attempt to execute **SQLTablePrivileges** on an updatable (keyset-driven or dynamic) returns SQL_SUCCESS_WITH_INFO indicating the cursor type has been changed.

The SQL Server ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *CatalogName* parameter: *Linked_Server_Name.Catalog_Name*.

See Also

<u>sp_table_privileges</u>

SQLTables

When restricted to the current database, **SQLTables** executes the Transact-SQL procedure **sp_tables** to report table catalog data for Microsoft® SQL Server[™].

The following table shows **SQLTables** parameter mapping for **sp_tables** stored procedure execution.

SQLTables parameter name	sp_tables parameter name
CatalogName	table_qualifier
SchemaName	table_owner
TableName	table_name
TableType	table_type

SQLTables can be executed on a static server cursor. An attempt to execute **SQLTables** on an updatable (dynamic or keyset) cursor will return SQL_SUCCESS_WITH_INFO indicating that the cursor type has been changed.

SQLTables reports tables from all databases when the *CatalogName* parameter is SQL_ALL_CATALOGS and all other parameters contain default values (NULL pointers). **SQLTables** does not make use of **sp_tables** in this special case.

To report available catalogs, schemas, and table types, **SQLTables** makes special use of empty strings (zero-length byte pointers). Empty strings are not default values (NULL pointers).

The SQL Server ODBC driver supports reporting information for tables on linked servers by accepting a two-part name for the *CatalogName* parameter: *Linked_Server_Name.Catalog_Name*.

SQLTables returns information about any tables whose names match *TableName* and are owned by the current user.

Example

// Get a list of all tables in the current database.
SQLTables(hstmt, NULL, 0, NULL, 0, NULL, 0, NULL, 0);

// Get a list of all tables in all databases.

SQLTables(hstmt, (SQLCHAR*) "%", SQL_NTS, NULL, 0, NULL, 0 // Get a list of databases on the current connection's server.

SQLTables(hstmt, (SQLCHAR*) "%", SQL_NTS, (SQLCHAR*)"", 0, 0, NULL, 0);

See Also

<u>sp_tables</u>

SQL Server Driver Extensions

The Microsoft® SQL Server[™] ODBC driver implements driver-specific functions to allow ODBC applications access to the bulk copy feature of SQL Server. The driver also has two driver-specific functions that allow ODBC applications to list the linked servers defined in a server, and then query the catalog of the linked servers.

Bulk-Copy Functions

The Microsoft® SQL Server[™]-specific bulk-copy API extension allows client applications to rapidly add data rows to, or extract data rows from, a SQL Server table.

See Also

Performing Bulk Copy Operations

bcp_batch

Commits all rows previously bulk copied from program variables and sent to Microsoft® SQL Server[™] by **bcp_sendrow**.

Syntax

DBINT bcp_batch (HDBC hdbc);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

Returns

The number of rows saved after the last call to **bcp_batch**, or **-**1 in case of error.

Remarks

Bulk copy batches define transactions. When an application uses **bcp_bind** and **bcp_sendrow** to bulk copy rows from program variables to SQL Server tables, the rows are committed only when the program calls **bcp_batch** or **bcp_done**.

You can call **bcp_batch** once every *n* rows or when there is a lull in incoming data (as in a telemetry application). If an application does not call **bcp_batch** the bulk copied rows are committed only when **bcp_done** is called.

See Also

<u>bcp_bind</u> <u>bcp_done</u> <u>bcp_sendrow</u>

bcp_bind

Binds data from a program variable to a table column for bulk copy into Microsoft® SQL ServerTM.

Syntax

RETCODE bcp_bind (HDBC hdbc, LPCBYTE pData, INT cbIndicator, DBINT cbData, LPCBYTE pTerm, INT cbTerm, INT eDataType, INT idxServerCol);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

pData

Is a pointer to the data copied. If *eDataType* is SQLTEXT, SQLNTEXT, or SQLIMAGE, *pData* can be NULL. A NULL *pData* indicates that long data values will be sent to SQL Server in chunks using **bcp_moretext**.

If indicators are present in the data, they appear in memory directly before the data. The *pData* parameter points to the indicator variable in this case, and the width of the indicator, the *cbIndicator* parameter, is used by bulk copy to address user data correctly.

cbIndicator

Is the length, in bytes, of a length or null indicator for the column's data. Valid indicator length values are 0 (when using no indicator), 1, 2, or 4.

Indicators appear in memory directly before any data. For example, the following structure type definition could be used to insert integer values into an SQL Server table using bulk copy:

typedef struct tagBCPBOUNDINT

{
int iIndicator;
int iValue;
} BCPBOUNDINT;

In the example case, the *pData* parameter would be set to the address of a declared instance of the structure, the address of the BCPBOUNDINT *iIndicator* structure member. The *cbIndicator* parameter would be set to the size of an integer (sizeof(int)), and the *cbData* parameter would again be set to the size of an integer (sizeof(int)). To bulk copy a row to the server containing a NULL value for the bound column, the value of the instance's *iIndicator* member should be set to SQL_NULL_DATA.

cbData

Is the count of bytes of data in the program variable, not including the length of any length or null indicator or terminator.

Setting *cbData* to SQL_NULL_DATA signifies that all rows copied to the server contain a NULL value for the column.

Setting *cbData* to SQL_VARLEN_DATA indicates that the system will use a string terminator, or other method, to determine the length of data copied.

For fixed-length data types, such as integers, the data type indicates the length of the data to the system. Therefore, for fixed-length data types, *cbData* can safely be SQL_VARLEN_DATA or the length of the data.

For SQL Server character and binary data types, *cbData* can be SQL_VARLEN_DATA, SQL_NULL_DATA, some positive value, or 0. If *cbData* is SQL_VARLEN_DATA, the system uses either a length/null indicator (if present) or a terminator sequence to determine the length of the data. If both are supplied, the system uses the one that results in the least amount of data being copied. If *cbData* is SQL_VARLEN_DATA, the data type of the column is an SQL Server character or binary type, and neither a

length indicator nor a terminator sequence is specified, the system returns an error message.

If *cbData* is 0 or a positive value, the system uses *cbData* as the data length. However, if, in addition to a positive *cbData* value, a length indicator or terminator sequence is provided, the system determines the data length by using the method that results in the least amount of data being copied.

The *cbData* parameter value represents the count of bytes of data. If character data is represented by Unicode wide characters, then a positive *cbData* parameter value represents the number of characters multiplied by the size in bytes of each character.

pTerm

Is a pointer to the byte pattern, if any, that marks the end of this program variable. For example, ANSI and MBCS C strings usually have a 1-byte terminator (\0).

If there is no terminator for the variable, set *pTerm* to NULL.

You can use an empty string ("") to designate the C null terminator as the program-variable terminator. Because the null-terminated empty string constitutes a single byte (the terminator byte itself), set *cbTerm* to 1. For example, to indicate that the string in *szName* is null-terminated and that the terminator should be used to indicate the length:

bcp_bind(hdbc, szName, 0, SQL_VARLEN_DATA, "", 1, SQLCHA

A nonterminated form of this example could indicate that 15 characters be copied from the *szName* variable to the second column of the bound table:

bcp_bind(hdbc, szName, 0, 15, NULL, 0, SQLCHARACTER, 2)

The bulk copy API performs Unicode-to-MBCS character conversion as required. Make sure that both the terminator byte string and the length of the byte string are set correctly. For example, to indicate that the string in *szName* is a Unicode wide character string, terminated by the Unicode null terminator value:

bcp_bind(hdbc, szName, 0, SQL_VARLEN_DATA, L''', sizeof(WCHAR), SQLNCHAR, 2)

If the bound SQL Server column is wide character, no conversion is performed on **bcp_sendrow**. If the SQL Server column is an MBCS character type, wide character to multibyte character conversion is performed as the data is sent to the SQL Server.

cbTerm

Is the count of bytes present in the terminator for the program variable, if any. If there is no terminator for the variable, set *cbTerm* to 0.

eDataType

Is the C data type of the program variable. The data in the program variable is converted to the type of the database column. If this parameter is 0, no conversion is performed.

For more information about a list of supported conversions, see the *ODBC 3.0 Programmer's Reference*.

The *eDataType* parameter is enumerated by the SQL Server data type tokens in Odbcss.h, not the ODBC C data type enumerators. For example, you can specify a two-byte integer, ODBC type SQL_C_SHORT, using the SQL Server-specific type SQLINT2.

idxServerCol

Is the ordinal position of the column in the database table to which the data is copied. The first column in a table is column 1. The ordinal position of a column is reported by **SQLColumns**.

Returns

SUCCEED or FAIL.

Remarks

Use **bcp_bind** for a fast, efficient way to copy data from a program variable into

a table in SQL Server.

Call **bcp_init** before calling this or any other bulk-copy function. Calling **bcp_init** sets the SQL Server target table for bulk copy. When calling **bcp_init** for use with **bcp_bind** and **bcp_sendrow**, the **bcp_init** *szDataFile* parameter, indicating the data file, is set to NULL; the **bcp_init** *eDirection* parameter is set to DB_IN.

Make a separate **bcp_bind** call for every column in the SQL Server table into which you want to copy. After the necessary **bcp_bind** calls have been made, then call **bcp_sendrow** to send a row of data from your program variables to SQL Server.

Whenever you want SQL Server to commit the rows already received, call **bcp_batch**. For example, call **bcp_batch** once for every 1000 rows inserted or at any other interval.

When there are no more rows to be inserted, call **bcp_done**. Failure to do so results in an error.

Control parameter settings, specified with **bcp_control**, have no effect on **bcp_bind** row transfers.

Calling **bcp_columns** when using **bcp_bind** results in an error.

Example

...
// Variables like henv not specified.
HDBC hdbc;
char szCompanyName[MAXNAME];
DBINT idCompany;
DBINT nRowsProcessed;
DBBOOL bMoreData;
char* pTerm = "\t\t";

// Application initiation, get an ODBC environment handle, allocate th
// hdbc, and so on.

•••

// Enable bulk copy prior to connecting on allocated hdbc. SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC SQL_IS_INTEGER);

```
// Connect to the data source; return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
 _T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
 {
 // Raise error and return.
 return;
 }
// Initialize bcp.
if (bcp_init(hdbc, "comdb..accounts_info", NULL, NULL
 DB IN = FAIL
 {
 // Raise error and return.
 return;
 }
// Bind program variables to table columns.
if (bcp_bind(hdbc, (LPCBYTE) &idCompany, 0, sizeof(DBINT), NUI
 SQLINT4, 1) == FAIL)
 ł
 // Raise error and return.
 return;
 }
if (bcp_bind(hdbc, (LPCBYTE) szCompanyName, 0, SQL_VARLEN
 (LPCBYTE) pTerm, strlen(pTerm), SQLCHARACTER, 2) == FAIL
 ł
 // Raise error and return.
 return;
```

```
}
while (TRUE)
  ł
 // Retrieve and process program data.
 if ((bMoreData = getdata(&idCompany, szCompanyName)) == TRU
   {
   // Send the data.
   if (bcp_sendrow(hdbc) == FAIL)
     ł
     // Raise error and return.
     return;
     }
    }
 else
    {
   // Break out of loop and carry on.
   break;
   }
  }
// Terminate the bulk copy operation.
if ((nRowsProcessed = bcp_done(hdbc)) == -1)
  {
 printf("Bulk-copy unsuccessful.\n");
 return;
 }
printf("%ld rows copied.\n", nRowsProcessed);
```

// Carry on.

•••

See Also

bcp_batch

bcp_colfmt

<u>bcp_collen</u>

<u>bcp_colptr</u>

bcp_columns

bcp_control

bcp_done

bcp_exec

<u>bcp_init</u>

bcp_moretext

bcp_sendrow

<u>SQLColumns</u>

bcp_colfmt

Specifies the source or target format of the data in a user file. When used as a source format, **bcp_colfmt** specifies the format of an existing data file used as the source of data in a bulk copy to a Microsoft® SQL Server[™] table. When used as a target format, the data file is created using the column formats specified with **bcp_colfmt**.

Syntax

RETCODE bcp_colfmt (HDBC hdbc, INT idxUserDataCol, BYTE eUserDataType, INT cbIndicator, DBINT cbUserData, LPCBYTE pUserDataTerm, INT cbUserDataTerm, INT idxServerCol);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

idxUserDataCol

Is the ordinal column number in the user data file for which the format is being specified. The first column is 1.

eUserDataType

Is the data type of this column in the user file. If different from the data type of the corresponding column in the database table (*idxServerColumn*), bulk copy converts the data if possible. For more information about supported data conversions, see the *ODBC 3.0 Programmer's Reference*.

The *eUserDataType* parameter is enumerated by the SQL Server data type tokens in Odbcss.h, not the ODBC C data type enumerators. For example, you can specify a character string, ODBC type SQL_C_CHAR, using the SQL Server-specific type SQLCHARACTER.

To specify the default data representation for the SQL Server data type, set this parameter to 0.

For a bulk copy out of SQL Server into a file, when *eUserDataType* is SQLDECIMAL or SQLNUMERIC:

- If the source column is not **decimal** or **numeric**, the default precision and scale are used.
- If the source column is **decimal** or **numeric**, the precision and scale of the source column are used.

cbIndicator

Is the length, in bytes, of a length/null indicator within the column data. Valid indicator length values are 0 (when using no indicator), 1, 2, or 4.

To specify default bulk copy indicator usage, set this parameter to SQL_VARLEN_DATA.

Indicators appear in memory directly before any data, and in the data file directly before the data to which they apply.

If more than one means of specifying a data file column length is used (such as an indicator and a maximum column length, or an indicator and a terminator sequence), bulk copy chooses the one that results in the least amount of data being copied.

Data files generated by bulk copy when no user intervention adjusts the format of the data contain indicators when the column data can vary in length or the column can accept NULL as a value.

cbUserData

Is the maximum length, in bytes, of this column's data in the user file, not including the length of any length indicator or terminator.

Setting *cbUserData* to SQL_NULL_DATA indicates that all values in the data file column are, or should be set to NULL.

Setting *cbUserData* to SQL_VARLEN_DATA indicates that the system should determine the length of data in each column. For some columns, this could mean that a length/null indicator is generated to precede data on a copy from SQL Server, or that the indicator is expected in data copied to SQL Server.

For SQL Server character and binary data types, *cbUserData* can be SQL_VARLEN_DATA, SQL_NULL_DATA, 0, or some positive value. If *cbUserData* is SQL_VARLEN_DATA, the system uses either the length indicator, if present, or a terminator sequence to determine the length of the data. If both a length indicator and a terminator sequence are supplied, bulk copy uses the one that results in the least amount of data being copied. If *cbUserData* is SQL_VARLEN_DATA, the data type is an SQL Server character or binary type, and neither a length indicator nor a terminator sequence is specified, the system returns an error message.

If *cbUserData* is 0 or a positive value, the system uses *cbUserData* as the maximum data length. However, if, in addition to a positive *cbUserData*, a length indicator or terminator sequence is provided, the system determines the data length by using the method that results in the least amount of data being copied.

The *cbUserData* value represents the count of bytes of data. If character data is represented by Unicode wide characters, then a positive *cbUserData* parameter value represents the number of characters multiplied by the size, in bytes, of each character.

pUserDataTerm

Is the terminator sequence to be used for this column. This parameter is useful mainly for character data types because all other types are of fixed length or, in the case of binary data, require an indicator of length to accurately record the number of bytes present.

To avoid terminating extracted data, or to indicate that data in a user file is not terminated, set this parameter to NULL.

If more than one means of specifying a user-file column length is used (such

as a terminator and a length indicator, or a terminator and a maximum column length), bulk copy chooses the one that results in the least amount of data being copied.

The bulk copy API performs Unicode-to-MBCS character conversion as required. Care must be taken to ensure that both the terminator byte string and the length of the byte string are set correctly.

cbUserDataTerm

Is the length, in bytes, of the terminator sequence to be used for this column. If no terminator is present or desired in the data, set this value to 0.

idxServerCol

Is the ordinal position of the column in the database table. The first column number is 1. The ordinal position of a column is reported by **SQLColumns**.

If this value is 0, bulk copy ignores the column in the data file.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_colfmt** function allows you to specify the user-file format for bulk copies. For bulk copy, a format contains the following parts:

- A mapping from user-file columns to database columns.
- The data type of each user-file column.
- The length of the optional indicator for each column.
- The maximum length of data per user-file column.
- The optional terminating byte sequence for each column.

• The length of the optional terminating byte sequence.

Each call to **bcp_colfmt** specifies the format for one user-file column. For example, to change the default settings for three columns in a five-column user data file, first call **bcp_columns(5)**, and then call **bcp_colfmt** five times, with three of those calls setting your custom format. For the remaining two calls, set *eUserDataType* to 0, and set *cbIndicator*, *cbUserData*, and *cbUserDataTerm* to 0, SQL_VARLEN_DATA, and 0 respectively. This procedure copies all five columns, three with your customized format and two with the default format.

The **bcp_columns** function must be called before any calls to **bcp_colfmt**.

You must call **bcp_colfmt** once for each column in the user file.

Calling **bcp_colfmt** more than once for any user-file column causes an error.

You do not need to copy all data in a user file to the SQL Server table. To skip a column, specify the format of the data for the column, setting the *idxServerCol* parameter to 0. If you want to skip a column, you must specify its type.

The **bcp_writefmt** function can be used to persist the format specification.

See Also

bcp_batch bcp_bind bcp_collen bcp_colptr bcp_columns bcp_control bcp_done bcp_exec bcp_init bcp_sendrow bcp_writefmt **SQLColumns**

bcp_collen

Sets the program variable data length for the current bulk copy into Microsoft® SQL ServerTM.

Syntax

```
RETCODE bcp_collen (
HDBC hdbc,
DBINT cbData,
INT idxServerCol );
```

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

cbData

Is the length of the data in the program variable, not including the length of any length indicator or terminator. Setting *cbData* to SQL_NULL_DATA indicates all rows copied to the server contain a NULL value for the column. Setting it to SQL_VARLEN_DATA indicates a string terminator or other method is used to determine the length of data copied. If both a length indicator and a terminator exist, the system uses the one that results in the least amount of data being copied.

idxServerCol

Is the ordinal position of the column in the table to which the data is copied. The first column is 1. The ordinal position of a column is reported by **SQLColumns**.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_collen** function allows you to change the program variable data length for a particular column when copying data to SQL Server with **bcp_sendrow**.

Initially, the program variable data length is determined when **bcp_bind** is called. If the program variable data length changes between calls to **bcp_sendrow** and no length prefix or terminator is being used, you can call **bcp_collen** to reset the length. The next call to **bcp_sendrow** uses the length set by the call to **bcp_collen**.

You must call **bcp_collen** once for each column in the table whose data length you want to modify.

See Also

bcp_sendrow

SQLColumns

bcp_colptr

Sets the program variable data address for the current copy into Microsoft® SQL Server[™].

Syntax

```
RETCODE bcp_colptr (
HDBC hdbc,
LPCBYTE pData,
INT idxServerCol );
```

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

pData

Is a pointer to the data to copy. If the bound data type is SQLTEXT, SQLNTEXT, or SQLIMAGE, *pData* can be NULL. A NULL *pData* indicates long data values will be sent to SQL Server in chunks using **bcp_moretext**.

idxServerCol

Is the ordinal position of the column in the database table to which the data is copied. The first column in a table is column 1. The ordinal position of a column is reported by **SQLColumns**.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_colptr** function allows you to change the address of source data for a

particular column when copying data to SQL Server with **bcp_sendrow**.

Initially, the pointer to user data is set by a call to **bcp_bind**. If the program variable data address changes between calls to **bcp_sendrow**, you can call **bcp_colptr** to reset the pointer to the data. The next call to **bcp_sendrow** sends the data addressed by the call to **bcp_colptr**.

There must be a separate **bcp_colptr** call for every column in the table whose data address you want to modify.

See Also

bcp_bind bcp_collen bcp_moretext bcp_sendrow SQLColumns

bcp_columns

Sets the total number of columns found in the user file for use with a bulk copy into or out of Microsoft® SQL Server[™].

Syntax

RETCODE bcp_columns (HDBC hdbc, INT nColumns);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

nColumns

Is the total number of columns in the user file. Even if you are preparing to bulk copy data from the user file to an SQL Server table and do not intend to copy all columns in the user file, you must still set *nColumns* to the total number of user-file columns.

Returns

SUCCEED or FAIL.

Remarks

This function can be called only after **bcp_init** has been called with a valid file name.

You should call this function only if you intend to use a user-file format that differs from the default. For more information about a description of the default user-file format, see <u>bcp_init</u>.

After calling **bcp_columns**, you must call **bcp_colfmt** for each column in the

user file to completely define a custom file format.

See Also

bcp_colfmt

bcp_control

Changes the default settings for various control parameters for a bulk copy between a file and Microsoft[®] SQL Server[™].

Syntax

RETCODE bcp_control (HDBC hdbc, INT eOption, void* iValue);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

eOption

Is one of the following:

BCP6xFILEFMT

When *iValue* is TRUE, specifies that columns stored in a data file use the SQL Server version 6.*x* format instead of the SQL Server 7.0 format.

Version 6.*x* format does not support several data types when bulk copying out from an SQL Server 7.0 database. Nullable **bit** values are converted to 0. **char**, **varchar**, **binary**, and **varbinary** values longer than 255 bytes are trunctated. **uniqueidentifier**, **nchar**, **nvarchar**, and **ntext** columns are not supported. Zero length data is converted to NULL.

When bulk copying in from a character data file column, blank input values are converted to NULL when *iValue* is set to FALSE and as follows when *iValue* is TRUE.

Target column data type	Resulting value
Any data type in the numeric category	0

binary or varbinary	0x00
datetime or smalldatetime	NULL
uniqueidentifier	NULL

When bulk copying in from a character data file column containing **datetime** strings, all **datetime** string formats supported by earlier DB-Library versions of bulk copy are supported.

When *iValue* is set to TRUE, a prefix of 0x is allowed for binary values specified in character mode data files. The prefix is not allowed if *iValue* is FALSE.

When iValue is set to FALSE, zero length indicates are stored as 0x00 in character mode data files and as 0x0000 in BCPUNICODE files.

BCPABORT

Stops a bulk-copy operation that is already in progress. Call **bcp_control** with an *eOption* of BCPABORT from another thread to stop a running bulk-copy operation. The *iValue* parameter is ignored.

BCPBATCH

Is the number of rows per batch. The default is 0, which indicates either all rows in a table, when data is being extracted, or all rows in the user data file, when data is being copied to an SQL Server. A value less than 1 resets BCPBATCH to the default.

BCPFILECP

iValue contains the number of the code page for the data file. You can specify the number of the code page, such as 1252 or 850, or one of these values:

BCPFILE_ACP: data in the file is in the Microsoft Windows® code page of the client.

BCPFILE_OEMCP: data in the file is in the OEM code page of the client (default).

BCPFILE_RAW: data in the file is in the code page of the SQL Server.

BCPFIRST

Is the first row of data to file or table to copy. The default is 1; a value less than 1 resets this option to its default.

BCPHINTS

iValue contains an SQLTCHAR character string pointer. The string addressed specifies either SQL Server bulk-copy processing hints or a Transact-SQL statement that returns a result set. If a Transact-SQL statement is specified that returns more than one result set, all result sets after the first are ignored. For more information about bulk-copy processing hints, see <u>bcp Utility</u>.

BCPKEEPIDENTITY

When *iValue* is TRUE, specifies that bulk copy functions insert data values supplied for SQL Server columns defined with an identity constraint. The input file must supply values for the identity columns. If this is not set, new identity values are generated for the inserted rows. Any data present in the file for the identity columns is ignored.

BCPKEEPNULLS

Specifies whether empty data values in the file will be converted to NULL values in the SQL Server table. When *iValue* is TRUE, empty values will be converted to NULL in the SQL Server table. The default is for empty values to be converted to a default value for the column in the SQL Server table if a default exists.

BCPLAST

Is the last row to copy. The default is to copy all rows; a value less than 1 resets this option to its default.

BCPMAXERRS

Is the number of errors allowed before the bulk copy operation fails. The default is 10; a value less than 1 resets this option to its default. Bulk copy imposes a maximum of 65,535 errors. An attempt to set this option to a value larger than 65,535 results in the option being set to 65,535.

BCPODBC

When TRUE, specifies that **datetime** and **smalldatetime** values saved in character format will use the ODBC timestamp escape sequence prefix and

suffix. The BCPODBC option only applies to BCP_OUT.

When FALSE, a **datetime** value representing January 1, 1997 is converted to the character string: 1997-01-01 00:00:00.000. When TRUE, the same **datetime** value is represented as: {ts '1997-01-01 00:00:00.000'}.

BCPUNICODEFILE

When TRUE, specifies the input file is a Unicode file.

FIRE_TRIGGERS

Specifies that INSERT and INSTEAD OF triggers defined on the destination table are fired once for each bulk copy batch. The inserted table passed to each trigger contains all of the rows inserted by the batch. Bulk copy operations that would otherwise be logged minimally are fully logged when FIRE_TRIGGERS is specified. No result sets generated by the triggers are returned to the client performing the bulk copy operation. Specify FIRE_TRIGGERS only when all of the INSERT and INSTEAD OF triggers on the destination table support multiple row inserts. The *iValue* parameter is ignored.

iValue

Is the value for the specified *eOption*. *iValue* is an integer value cast to a void pointer to allow for future expansion to 64 bit values.

Returns

SUCCEED or FAIL.

Remarks

This function sets various control parameters for bulk-copy operations, including the number of errors allowed before canceling a bulk copy, the numbers of the first and last rows to copy from a data file, and the batch size.

This function is also used to specify the SELECT statement when bulk copying out from SQL Server the result set of a SELECT. Set *eOption* to BCPHINTS and set *iValue* to have a pointer to an SQLTCHAR string containing the SELECT statement.

These control parameters are only meaningful when copying between a user file and an SQL Server table. Control parameter settings have no effect on rows copied to SQL Server with **bcp_sendrow**.

Example

```
...
// Variables like henv not specified.
SQLHDBC hdbc;
DBINT nRowsProcessed;
```

// Application initiation, get an ODBC environment handle, allocate th
// hdbc, and so on.

•••

```
// Enable bulk copy prior to connecting on allocated hdbc.
SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC
SQL_IS_INTEGER);
```

```
// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
_T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
{
// Raise error and return.
return;
}
// Initialize bulk copy.
if (bcp_init(hdbc, _T("address"), _T("address.add"), _T("addr.err"),
DB_IN) == FAIL)
{
// Raise error and return.
return;
}
```

```
// Set the number of rows per batch.
if (bcp_control(hdbc, BCPBATCH, (void*) 1000) == FAIL)
 // Raise error and return.
 return;
 }
// Set file column count.
if (bcp_columns(hdbc, 1) == FAIL)
  {
 // Raise error and return.
 return;
 }
// Set the file format.
if (bcp_colfmt(hdbc, 1, 0, 0, SQL_VARLEN_DATA, '\n', 1, 1)
 == FAIL)
  {
 // Raise error and return.
 return;
 }
// Execute the bulk copy.
if (bcp_exec(hdbc, &nRowsProcessed) == FAIL)
  {
 // Raise error and return.
 return;
  }
```

printf("%ld rows processed by bulk copy.", nRowsProcessed);

See Also

bcp_exec

bcp_sendrow

bcp_done

Ends a bulk copy from program variables to Microsoft® SQL Server[™] performed with **bcp_sendrow**.

Syntax

DBINT bcp_done (HDBC *hdbc*);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

Returns

The number of rows permanently saved after the last call to **bcp_batch** or **-**1 in case of error.

Remarks

Call **bcp_done** after the last call to **bcp_sendrow** or **bcp_moretext**. Failure to call **bcp_done** after copying all data results in errors.

See Also

bcp_batch

bcp_moretext

bcp_sendrow

bcp_exec

Executes a complete bulk copy of data between a database table and a user file.

Syntax

RETCODE bcp_exec (HDBC *hdbc*, **LPDBINT** *pnRowsProcessed*);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

pnRowsProcessed

Is a pointer to a DBINT. The **bcp_exec** function fills this DBINT with the number of rows successfully copied. If *pnRowsProcessed* is NULL, it is ignored by **bcp_exec**.

Returns

SUCCEED, SUCCEED_ASYNC, or FAIL. The **bcp_exec** function returns SUCCEED if all rows are copied. **bcp_exec** returns SUCCEED_ASYNC if an asynchronous bulk copy operation is still outstanding. **bcp_exec** returns FAIL if a complete failure occurs, or if the number of rows generating errors reaches the value specified for BCPMAXERRS using **bcp_control**. BCPMAXERRS defaults to 10. The BCPMAXERRS option affects only the syntax errors detected by the provider while reading the rows from the data file (and not the rows sent to the server). Server aborts the batch when it detects an error with a row. Check the *pnRowsProcessed* parameter for the number of rows successfully copied.

Remarks

This function copies data from a user file to a database table or vice versa, depending on the value of the *eDirection* parameter in **bcp_init**.

Before calling **bcp_exec**, call **bcp_init** with a valid user file name. Failure to do so results in an error.

bcp_exec is the only bulk copy function that is likely to be outstanding for any length of time. It is therefore the only bulk copy function that supports asynchronous mode. To set asynchronous mode, use **SQLSetConnectAttr** to set SQL_ATTR_ASYNC_ENABLE to SQL_ASYNC_ENABLE_ON before calling **bcp_exec**. To test for completion, call **bcp_exec** with the same parameters. If the bulk copy has not yet completed, **bcp_exec** returns SUCCEED_ASYNC. It also returns in *pnRowsProcessed* a status count of the number of rows that have been sent to the server. Rows sent to the server are not committed until the end of a batch has been reached.

Example

The following example shows how to use **bcp_exec**:

... // Variables like henv not specified. HDBC hdbc; DBINT nRowsProcessed;

// Application initiation, get an ODBC environment handle, allocate the
// hdbc, and so on.

•••

// Enable bulk copy prior to connecting on allocated hdbc. SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC SQL_IS_INTEGER);

```
// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
    _T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
{
```

```
// Raise error and return.
 return;
 }
// Initialize bulk copy.
if (bcp_init(hdbc, _T("pubs..authors"), _T("authors.sav"), NULL, DB_
 == FAIL)
  {
 // Raise error and return.
 return;
 }
// Now, execute the bulk copy.
if (bcp_exec(dbproc, &nRowsProcessed) == FAIL)
  ł
 if (nRowsProcessed == -1)
   printf("No rows processed on bulk copy execution.\n");
   }
 else
    {
   printf("Incomplete bulk copy. Only %ld row%s copied.\n",
     nRowsProcessed, (nRowsProcessed == 1) ? "": "s");
    }
 return;
 }
printf("%ld rows processed.\n", nRowsProcessed);
```

// Carry on.

•••

See Also

<u>bcp_init</u>

bcp_getcolfmt

Used to find the column format property value.

Syntax

RETCODE bcp_getcolfmt (HDBC hdbc, INT field, INT property, void* pValue, INT cbvalue, INT* pcbLen);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

field

Is the column number for which the property is retrieved.

property

Is one of the property constants.

pValue

Is the pointer to the buffer in which to retrieve the property value.

cbValue

Is the length of the property buffer in bytes.

pcbLen

Pointer to length of the data that is being returned in the property buffer.

Returns

SUCCEED or FAIL.

Remarks

Column format property values are listed in the **bcp_setcolfmt** topic. The column format property values are set by calling the **bcp_setcolfmt** function, and the **bcp_getcolfmt** function is used to find the column format property value.

See Also

bcp_setcolfmt

bcp_init

Initializes bulk copy operation.

Syntax

RETCODE bcp_init (HDBC hdbc, LPCTSTR szTable, LPCTSTR szDataFile, LPCTSTR szErrorFile, INT eDirection);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

szTable

Is the name of the database table to be copied into or out of. This name can also include the database name or the owner name. For example, **pubs.gracie.titles**, **pubs..titles**, **gracie.titles**, and **titles** are all legal table names.

If *eDirection* is DB_OUT, *szTable* can also be the name of a database view.

If *eDirection* is DB_OUT and a SELECT statement is specified using **bcp_control** before **bcp_exec** is called, **bcp_init** *szTable* must be set to NULL.

szDataFile

Is the name of the user file to be copied into or out of. If data is being copied directly from variables by using **bcp_sendrow**, set *szDataFile* to NULL.

szErrorFile

Is the name of the error file to be filled with progress messages, error

messages, and copies of any rows that, for any reason, could not be copied from a user file to a table. If NULL is passed as *szErrorFile*, no error file is used.

eDirection

Is the direction of the copy, either DB_IN or DB_OUT. DB_IN indicates a copy from program variables or a user file to a table. DB_OUT indicates a copy from a database table to a user file. You must specify a user file name with DB_OUT.

Returns

SUCCEED or FAIL.

Remarks

Call **bcp_init** before calling any other bulk-copy function. **bcp_init** performs the necessary initializations for a bulk copy of data between the workstation and Microsoft® SQL Server[™].

The **bcp_init** function must be provided with an ODBC connection handle enabled for use with bulk copy functions. To enable the handle, use **SQLSetConnectAttr** with SQL_COPT_SS_BCP set to SQL_BCP_ON on an allocated, but not connected, connection handle. Attempting to assign the attribute on a connected handle results in an error.

When a data file is specified, **bcp_init** examines the structure of the database source or target table, not the data file. **bcp_init** specifies data format values for the data file based on each column in the database table, view, or SELECT result set. This specification includes the data type of each column, the presence or absence of a length or null indicator and terminator byte strings in the data, and the width of fixed-length data types. **bcp_init** sets these values as follows:

• The data type specified is the data type of the column in the database table, view, or SELECT result set. The data type is enumerated by SQL Server native data types specified in Odbcss.h. Data itself is represented in its computer form. That is, data from a column of **integer** data type is represented by a four-byte sequence that is big-or little-endian based on the computer that created the data file.

- If a database data type is fixed in length, the data file data is also fixed in length. Bulk-copy functions that process data (for example, bcp_exec) parse data rows expecting the length of the data in the data file to be identical to the length of the data specified in the database table, view, or SELECT column list. For example, data for a database column defined as char(13) must be represented by 13 characters for each row of data in the file. Fixed-length data can be prefixed with a null indicator if the database column allows null values.
- When terminator-byte sequence is defined, the length of the terminatorbyte sequence is set to 0.
- When copying to SQL Server, the data file must have data for each column in the database table. When copying from SQL Server, data from all columns in the database table, view, or SELECT result set are copied to the data file.
- When copying to SQL Server, the ordinal position of a column in the data file must be identical to the ordinal position of the column in the database table. When copying from SQL Server, **bcp_exec** places data based on the ordinal position of the column in the database table.
- If a database data type is variable in length (for example, **varbinary(22)**) or if a database column can contain null values, data in the data file is prefixed by a length/null indicator. The width of the indicator varies based on the data type and version of bulk copy. The **bcp_control** option BCP6xFILEFMT provides compatibility between earlier bulk copy data files and servers running later versions of SQL Server by indicating when the width of indicators in the data is narrower than expected.

To change data format values specified for a data file, call **bcp_columns** and **bcp_colfmt**.

Bulk copies to SQL Server can be optimized for tables that do not contain indexes by setting the database option **select into/bulkcopy** (see the example). For more information, see <u>Optimizing Bulk Copy Performance</u>.

If no data file is used, you must call **bcp_bind** to specify the format and location in memory of the data for each column, then copy data rows to the SQL Server using **bcp_sendrow**.

Example

Setting the **select into/bulkcopy** option allows faster bulk copies for tables that do not contain indexes.

... // Variables like henv not specified. HDBC hdbc; SQLHSTMT hstmt;

// Application initiation, get an ODBC environment handle, allocate th
// hdbc, and so on.

•••

// Enable bulk copy prior to connecting on allocated hdbc. SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC SQL_IS_INTEGER);

```
// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
    _T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
    {
    // Raise error and return.
    return;
    }
```

// Get a statement handle and set the select into/bulkcopy database

```
// option to TRUE.
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if (!SQL_SUCCEEDED(SQLExecDirect(hstmt,
 _T("sp_dboption 'mydb', 'select into/bulkcopy', 'true'"),
 SQL_NTS)))
 {
 // Raise error and return.
 return;
 }
// Flush the statement handle.
while (SQL_SUCCEEDED(SQLMoreResults(hstmt)))
 ;
// Initialize bulk copy, perform copies, and so on.
...
// Turn off the select into/bulkcopy database option.
if (!SQL_SUCCEEDED(SQLExecDirect(hstmt,
 _T("sp_dboption 'mydb', 'select into/bulkcopy', 'false'"),
 SQL_NTS)))
 {
 // Raise error and return.
 return;
 }
// Carry on.
...
```

See Also

bcp_bind

bcp_control bcp_colfmt bcp_columns bcp_sendrow Logged and Minimally Logged Bulk Copy Operations SQLSetConnectAttr

bcp_moretext

Sends part of a long, variable-length data type value to Microsoft® SQL ServerTM.

Syntax

RETCODE bcp_moretext (HDBC hdbc, DBINT cbData, LPCBYTE pData);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

cbData

Is the number of bytes of data being copied to SQL Server from the data referenced by *pData*.

pData

Is a pointer to the supported, long, variable-length data chunk to be sent to SQL Server.

Returns

SUCCEED or FAIL.

Remarks

This function can be used in conjunction with **bcp_bind** and **bcp_sendrow** to copy long, variable-length data values to SQL Server in a number of smaller chunks. **bcp_moretext** can be used with columns that have SQL Server data types enumerated with SQLTEXT, SQLNTEXT, and SQLIMAGE only.

bcp_moretext does not support data conversions, the data supplied must match the data type of the target column.

If **bcp_bind** is called with a nonNULL *pData* parameter for data types that are supported by **bcp_moretext**, **bcp_sendrow** sends the entire data value, regardless of length. If, however, **bcp_bind** has a NULL *pData* parameter for supported data types, **bcp_moretext** can be used to copy data immediately after a successful return from **bcp_sendrow** indicating that any bound columns with data present have been processed.

If you use **bcp_moretext** to send one supported data type column in a row, you must also use it to send all other supported data type columns in the row.

Calling either **bcp_bind** or **bcp_collen** sets the total length of all data parts to be copied to the SQL Server column. An attempt to send SQL Server more bytes than specified in the call to **bcp_bind** or **bcp_collen** generates an error. This error would arise, for example, in an application which used **bcp_collen** to set the length of available data for an SQL Server **text** column to 4500, then called **bcp_moretext** five times while indicating on each call that the data buffer length was 1000 bytes long.

If a copied row contains more than one long, variable-length column, **bcp_moretext** first sends its data to the lowest ordinally numbered column, followed by the next lowest ordinally numbered column, and so on. Correct setting of the total length of expected data is important. There is no way to signal, outside of the length setting, that all data for a column has been received by bulk copy.

An application normally calls **bcp_sendrow** and **bcp_moretext** within loops to send a number of rows of data. Here's an outline of how to do this for a table containing two **text** columns:

```
while (there are still rows to send)
{
    bcp_collen(..., total length of data for first text column,
    first text column's ordinal position);
    bcp_collen(..., total length of data for second text column,
    second text column's ordinal position);
```

```
bcp_sendrow(...);
```

for (all the data in the first text column)
bcp_moretext(...);

```
for (all the data in the second text column)
bcp_moretext(...);
}
```

Example

This example shows how to use **bcp_moretext** with **bcp_bind** and **bcp_sendrow**.

...
// Variables like henv not specified.
HDBC hdbc;
DBINT idRow = 5;
char* pPart1 = "This text value isn't very long,";
char* pPart2 = " but it's broken into three parts";
char* pPart3 = " anyhow.";
DBINT cbAllParts;
DBINT nRowsProcessed;

// Application initiation, get an ODBC environment handle, allocate the
// hdbc, and so on.

•••

// Enable bulk copy prior to connecting on allocated hdbc. SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC SQL_IS_INTEGER);

// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT)

```
_T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
 // Raise error and return.
 return;
 }
// Initialize bulk copy.
if (bcp_init(hdbc, "comdb..articles", NULL, NULL, DB_IN) == FAIL)
 {
 // Raise error and return.
 return;
 }
// Bind program variables to table columns.
if (bcp_bind(hdbc, (LPCBYTE) &idRow, 0, SQL_VARLEN_DATA, N
 SQLINT4, 1) == FAIL)
 {
 // Raise error and return.
 return;
 }
cbAllParts = (DBINT) (strlen(pPart1) + strlen(pPart2) + strlen(pPart3)
if (bcp_bind(hdbc, NULL, 0, cbAllParts, NULL, 0, SQLTEXT, 2) == I
 {
 // Raise error and return.
 return;
 }
// Send this row, with the text value broken into three chunks.
if (bcp_sendrow(hdbc) == FAIL)
 {
 // Raise error and return.
 return;
```

}

```
if (bcp_moretext(hdbc, (DBINT) strlen(pPart1), pPart1) == FAIL)
{
    // Raise error and return.
    return;
    }
if (bcp_moretext(hdbc, (DBINT) strlen(pPart2), pPart2) == FAIL)
    {
        // Raise error and return.
        return;
    }
if (bcp_moretext(hdbc, (DBINT) strlen(pPart3), pPart3) == FAIL)
    {
        // Raise error and return.
        return;
    }
```

// All done. Get the number of rows processed (should be one).
nRowsProcessed = bcp_done(hdbc);

// Carry on.

See Also

```
bcp_bind
```

bcp_collen

bcp_sendrow

bcp_readfmt

Reads a data file format definition from the specified format file.

Syntax

RETCODE bcp_readfmt (HDBC *hdbc*, **LPCTSTR** *szFormatFile* **)**;

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

szFormatFile

Is the path and file name of the file containing the format values for the data file.

Returns

SUCCEED or FAIL.

Remarks

After **bcp_readfmt** reads the format values, it makes the appropriate calls to **bcp_columns** and **bcp_colfmt**. There is no need for you to parse a format file and make these calls.

To persist a format file, call **bcp_writefmt**. Calls to **bcp_readfmt** can reference saved formats. For more information, see <u>bcp_init</u>.

Alternately, the bulk-copy utility (**bcp**) can save user-defined data formats in files that can be referenced by **bcp_readfmt**. For more information about the **bcp** utility and the structure of **bcp** data format files, see <u>Using Format Files</u>.

Note The format file must have been produced by version 4.2 or later of the **bcp**

utility.

Example

// Variables like henv not specified.

HDBC hdbc;

DBINT nRowsProcessed;

// Application initiation, get an ODBC environment handle, allocate th
// hdbc, and so on.

•••

```
// Enable bulk copy prior to connecting on allocated hdbc.
SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC
SQL_IS_INTEGER);
```

```
// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
_T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS))))
{
// Raise error and return.
return;
}
// Initialize bulk copy.
if (bcp_init(hdbc, _T("myTable"), _T("myData.csv"),
_T("myErrors"), DB_IN) == FAIL)
{
// Raise error and return.
return;
}
if (bcp_readfmt(hdbc, _T("myFmtFile.fmt")) == FAIL)
{
```

```
// Raise error and return.
return;
}
if (bcp_exec(hdbc, &nRowsProcessed) == SUCCEED)
{
printf("%ld rows copied to SQL Server\n", nRowsProcessed);
}
```

```
// Carry on.
```

•••

See Also

bcp_colfmt

<u>bcp_columns</u>

bcp_writefmt

bcp_sendrow

Sends a row of data from program variables to Microsoft® SQL ServerTM.

Syntax

RETCODE bcp_sendrow (HDBC *hdbc*);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

Returns

SUCCEED or FAIL.

Remarks

The **bcp_sendrow** function builds a row from program variables and sends it to SQL Server.

Before calling **bcp_sendrow**, you must make calls to **bcp_bind** to specify the program variables containing row data.

If **bcp_bind** is called specifying a long, variable-length data type, for example, an *eDataType* parameter of SQLTEXT and a nonNULL *pData* parameter, **bcp_sendrow** sends the entire data value, just as it does for any other data type. If, however, **bcp_bind** has a NULL *pData* parameter, **bcp_sendrow** returns control to the application immediately after all columns with data specified are sent to SQL Server. The application can then call **bcp_moretext** repeatedly to send the long, variable-length data to SQL Server, a chunk at a time. For more information, see <u>bcp_moretext</u>.

When **bcp_sendrow** is used to bulk copy rows from program variables into SQL Server tables, rows are committed only when the user calls **bcp_batch** or **bcp_done**. The user can choose to call **bcp_batch** once every *n* rows or when

there is a lull between periods of incoming data. If **bcp_batch** is never called, the rows are committed when **bcp_done** is called.

See Also

bcp_batch bcp_bind bcp_done

bcp_setcolfmt

The **bcp_setcolfmt** function supercedes the **bcp_colfmt**. In specifying the column collation, **bcp_setcolfmt** function must be used.

This function provides a flexible approach to specifying the column format in a bulk copy operation. It is used to set individual column format attributes. Each call to **bcp_setcolfmt** will set one column format attribute.

The **bcp_setcolfmt** function specifies the source or target format of the data in a user file. When used as a source format, **bcp_setcolfmt** specifies the format of an existing data file used as a data source of data in a bulk copy to a table in Microsoft® SQL Server[™]. When used as a target format, the data file is created using the column formats specified with **bcp_setcolfmt**.

Syntax

RETCODE bcp_setcolfmt (

HDBC hdbc, INT field, INT property, void* pValue, INT cbValue);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

field

Is the ordinal column number for which the property is being set.

property

Is one of the property constants. Property constants are defined in this table.

Property	Value	Description
		I

BCP_FMT_TYPE	BYTE	Is the data type of this column in the user file. If different from the data type of the corresponding column in the database table, bulk copy converts the data if possible. For more information, see the ODBC documentation.
		The BCP_FMT_TYPE parameter is enumerated by the SQL Server data type tokens in Odbcss.h, rather than the ODBC C data type enumerators. For example, you can specify a character string, ODBC type SQL_C_CHAR, using the SQLCHARACTER type specific to SQL Server.
		To specify the default data representation for the SQL Server data type, set this parameter to 0.
		For a bulk copy out of SQL Server into a file, when BCP_FMT_TYPE is SQLDECIMAL or SQLNUMERIC:
		• If the source column is not decimal or numeric , the default precision and scale are used.
		• If the source column is

		decimal or numeric , the precision and scale of the source column are used.
BCP_FMT_INDICATOR_LEN	INT	Is the length in bytes of the indicator (prefix).
		It is the length, in bytes, of a length/null indicator within the column data. Valid indicator length values are 0 (when using no indicator), 1, 2, or 4.
		To specify default bulk copy indicator usage, set this parameter to SQL_VARLEN_DATA.
		Indicators appear in memory directly before any data, and in the data file directly before the data to which they apply.
		If more than one means of specifying a data file column length is used (such as an indicator and a maximum column length, or an indicator and a terminator sequence), bulk copy chooses the one that results in the least amount of data being copied.
		Data files generated by bulk copy when no user intervention adjusts the format of the data contain indicators when the column data can vary in length or the column can accept

		NULL as a value.
BCP_FMT_DATA_LEN	DBINT	Is the length in bytes of the data (column length)
		It is the maximum length, in bytes, of this column's data in the user file, not including the length of any length indicator or terminator.
		Setting BCP_FMT_DATA_LEN to SQL_NULL_DATA indicates that all values in the data file column are, or should be set to, NULL.
		Setting BCP_FMT_DATA_LEN to SQL_VARLEN_DATA indicates that the system should determine the length of data in each column. For some columns, this could mean that a length/null indicator is generated to precede data on a copy from SQL Server, or that the indicator is expected in data copied to SQL Server.
		For SQL Server character and binary data types, BCP_FMT_DATA_LEN can be SQL_VARLEN_DATA, SQL_NULL_DATA, 0, or some positive value. If BCP_FMT_DATA_LEN is SQL_VARLEN_DATA, the

system uses either the length indicator, if present, or a terminator sequence to determine the length of the data. If both a length indicator and a terminator sequence are supplied, bulk copy uses the one that results in the least amount of data being copied. If BCP FMT DATA LEN is SQL VARLEN DATA, the data type is an SQL Server character or binary type, and neither a length indicator nor a terminator sequence is specified, the system returns an error message.

If BCP_FMT_DATA_LEN is 0 or a positive value, the system uses BCP_FMT_DATA_LEN as the maximum data length. However, if, in addition to a positive

BCP_FMT_DATA_LEN, a length indicator or terminator sequence is provided, the system determines the data length by using the method that results in the least amount of data being copied.

The BCP_FMT_DATA_LEN value represents the count of bytes of data. If character data is represented by Unicode wide characters, then a positive BCP_FMT_DATA_LEN

		parameter value represents the number of characters multiplied by the size, in bytes, of each character.
BCP_FMT_TERMINATOR	LPCBYTE	Pointer to the terminator sequence (either ANSI or Unicode as appropriate) to be used for this column. This parameter is useful mainly for character data types because all other types are of fixed length or, in the case of binary data, require an indicator of length to accurately record the number of bytes present.
		To avoid terminating extracted data, or to indicate that data in a user file is not terminated, set this parameter to NULL.
		If more than one means of specifying a user-file column length is used (such as a terminator and a length indicator, or a terminator and a maximum column length), bulk copy chooses the one that results in the least amount of data being copied.
		The bulk copy API performs Unicode-to-MBCS character conversion as required. Care must be taken to ensure that both the terminator byte string and the length of the byte string are set correctly.

BCP_FMT_SERVER_COL	INT	Ordinal position of the column
		in the database
BCP_FMT_COLLATION	LPCSTR	Collation name.

pValue

Is the pointer to the value to associate to the *property*. It allows each column format property to be set individually.

cbvalue

Is the length of the property buffer in bytes.

Returns

SUCCEED or FAIL.

Remarks

This function supercedes the **bcp_colformat** function. All the functionality of **bcp_colformat** is provided in **bcp_setcolformat** function. In addition, support for column collation is also provided. It is recommended that the following column format attributes be set in the order given below:

BCP_FMT_SERVER_COL

BCP_FMT_DATA_LEN

BCP_FMT_TYPE

The **bcp_setcolfmt** function allows you to specify the user-file format for bulk copies. For bulk copy, a format contains the following parts:

- A mapping from user-file columns to database columns.
- The data type of each user-file column.
- The length of the optional indicator for each column.

- The maximum length of data per user-file column.
- The optional terminating byte sequence for each column.
- The length of the optional terminating byte sequence.

Each call to **bcp_setcolfmt** specifies the format for one user-file column. For example, to change the default settings for three columns in a five-column user data file, first call **bcp_columns(5)**, and then call **bcp_setcolfmt** five times, with three of those calls setting your custom format. For the remaining two calls, set BCP_FMT_TYPE to 0, and set BCP_FMT_INDICATOR_LENGTH, BCP_FMT_DATA_LEN, and cbValue to 0, SQL_VARLEN_DATA, and 0 respectively. This procedure copies all five columns, three with your customized format and two with the default format.

The **bcp_columns** function must be called before calling **bcp_setcolfmt**.

You must call **bcp_setcolfmt** once for each property of each column in the user file.

You do not need to copy all data in a user file to the SQL Server table. To skip a column, specify the format of the data for the column, setting the BCP_FMT_SERVER_COL parameter to 0. If you want to skip a column, you must specify its type.

The **bcp_writefmt** function can be used to persist the format specification.

See Also

bcp_getcolfmt

bcp_writefmt

Creates a format file containing a description of the format of the current bulk copy data file.

Syntax

RETCODE bcp_writefmt (HDBC *hdbc*, **LPCTSTR** *szFormatFile*);

Arguments

hdbc

Is the bulk copy-enabled ODBC connection handle.

szFormatFile

Is the path and file name of the user file to receive format values for the data file.

Returns

SUCCEED or FAIL.

Remarks

The format file specifies the data format of a data file created by bulk copy. Calls to **bcp_columns** and **bcp_colfmt** define the format of the data file. **bcp_writefmt** saves this definition in the file referenced by *szFormatFile*. For more information, see <u>bcp_init</u>.

For more information about the structure of **bcp** data format files, see <u>Using</u> <u>Format Files</u>.

To load a saved format file, use **bcp_readfmt**.

Note The format file produced by **bcp_writefmt** is supported only by versions

of the **bcp** utility distributed with Microsoft® SQL Server[™] version 7.0.

Example

// Variables like henv not specified.

HDBC hdbc;

DBINT nRowsProcessed;

// Application initiation, get an ODBC environment handle, allocate the
// hdbc, and so on.

•••

```
// Enable bulk copy prior to connecting on allocated hdbc.
SQLSetConnectAttr(hdbc, SQL_COPT_SS_BCP, (SQLPOINTER) SC
SQL_IS_INTEGER);
```

```
// Connect to the data source, return on error.
if (!SQL_SUCCEEDED(SQLConnect(hdbc, _T("myDSN"), SQL_NT
_T("myUser"), SQL_NTS, _T("myPwd"), SQL_NTS)))
{
    // Raise error and return.
    return;
    }
// Initialize bulk copy.
if (bcp_init(hdbc, _T("myTable"), _T("myData.csv"),
    _T("myErrors"), DB_OUT) == FAIL)
    {
    // Raise error and return.
    return;
    }
if (bcp_columns(hdbc, 3) == FAIL)
    {
```

```
// Raise error and return.
return;
```

```
}
```

```
bcp_colfmt(hdbc, 1, SQLCHARACTER, 0, SQL_VARLEN_DATA, '\1
bcp_colfmt(hdbc, 2, SQLCHARACTER, 0, SQL_VARLEN_DATA, '\1
bcp_colfmt(hdbc, 3, SQLCHARACTER, 0, SQL_VARLEN_DATA, '\1
```

```
if (bcp_writefmt(hdbc, _T("myFmtFile.fmt")) == FAIL)
  {
   // Raise error and return.
  return;
  }
```

```
if (bcp_exec(hdbc, &nRowsProcessed) == SUCCEED)
{
    printf("%ld rows copied from SQL Server\n", nRowsProcessed);
}
```

```
// Carry on.
```

•••

See Also

```
bcp_colfmt
bcp_columns
bcp_readfmt
```

Schema Functions Supporting Distributed Queries

The Microsoft® SQL Server[™] ODBC driver provides extended catalog functions that support SQL Server distributed queries.

The catalog functions **SQLLinkedCatalogs** and **SQLLinkedServers** list data sources available for distributed query. A linked server is an OLE DB data source. For the ODBC application, the name of the linked server can qualify tables and columns in a query. SQL Server distributes the query as required.

The extended functions are implemented in the SQL Server ODBC driver. The application developer links with Odbcbcp.lib to build an application by using **SQLLinkedCatalogs** or **SQLLinkedServers**.

In addition, the SQL Server ODBC driver supports using several standard ODBC functions to get catalog information for tables or linked servers.

See Also

Distributed Queries

Using Catalog Functions

SQLLinkedCatalogs

SQLLinkedCatalogs returns a list of catalogs available on a linked server.

Syntax

SQLRETURN SQLLinkedCatalogs(SQLHSTMT hstmt, SQLTCHAR* ServerName, SQLSMALLINT NameLength1)

Arguments

hstmt

Is an ODBC statement handle.

ServerName

Is the name of the linked server. Linked server names are returned in the SRV_NAME column of the result set defined for **SQLLinkedServers**.

NameLength1

Is the length of **ServerName*, in characters.

Returns

SUCCEED or FAIL.

Comments

Microsoft[®] SQL Server[™] linked servers are OLE DB data sources. Some OLE DB data sources expose catalogs. For those that do, **SQLLinkedCatalogs** returns the list of data source exposed catalogs.

SQLLinkedCatalogs returns a result set defined as follows.

	Column		
Column name	number	Data type	Comments

CATALOG_NAME	1	SQLWCHAR	Name of the catalog.
DESCRIPTION	2	SQLWCHAR	Human-readable
			description of the catalog.

SQLLinkedServers

SQLLinkedServers returns a list of data sources that can participate in distributed queries.

Syntax

SQLRETURN SQLLinkedServers(SQLHSTMT hstmt)

Arguments

hstmt

Is an ODBC statement handle.

Returns

SUCCEED or FAIL.

Comments

Microsoft® SQL Server[™] linked servers are OLE DB data sources that can be referenced using four-part names in distributed queries. **SQLLinkedServers** returns the properties defining the data source as a result set. Linked servers are defined to SQL Server using the **sp_addlinkedserver** system stored procedure. The columns of the result set are defined as follows.

	Column		
Column name	number	Data type	Comments
SRV_NAME	1	SQLWCHAR	Name of the linked
			server.
SRV_PROVIDERNAME	2	SQLWCHAR	Friendly name of the
			OLE DB provider for
			this linked server.
SRV_PRODUCT	3	SQLWCHAR	Product name for this
			linked server.

SRV_DATASOURCE	4	Name that identifies the source of the data to this provider.
SRV_PROVIDERSTRING	5	Provider-specific string that identifies the source of the data.
SRV_LOCATION	6	Location argument that identifies the database to this provider.
SRV_CAT	7	Name of the catalog containing the data on the linked server.

SRV_PROVIDERSTRING is NULL when the SRV_DATASOURCE and SRV_CAT information is sufficient to identify the source of the data to the provider.

When the linked server is a server running an instance of SQL Server, SRV_DATASOURCE is the name of the server, SRV_CATALOG is the name of the database, and SRV_LOCATION is NULL.

See Also

Distributed Queries

sp_addlinkedserver

ODBC Samples

The following samples have been developed for ODBC version 3.0 and later. The samples have all been developed with Microsoft® Visual C++® version 6.0, and some expose properties of the Microsoft Foundation Classes.

When you choose to install sample files, the ODBC 3.*x* samples are installed to the C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\Odbc directory.

All samples include a project file (.dsw extension) created by Visual C++ 6.0. The project file can be opened in Visual C++ 6.0.

To open the project file in Visual C++ 6.0 and compile it:

- 1. On the **File** menu, click **Open Workspace**.
- 2. In the **Files of type** box, click **Workspaces (*.dsw)**.
- 3. Click the project file name.
- 4. From the **Tools** menu, choose **Options**, and then click the **Directories** tab.
- 5. From the **Show directories for** box, choose Include files and Library files, and ensure that the following directories are included and appear at the top of list:
 - Include files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Include
 - Library files: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Lib
- 6. From the **Build** menu, choose **Rebuild All** or **Build** *.exe.

When the project file is opened, Visual C++ generates appropriate supporting files.

The default build configuration for all samples is Win32[®] Debug, which will build the samples as 32-bit applications.

Note To compile ODBC version 3.0 samples, you must obtain the ODBC 3.0 SDK available from Microsoft Press®, or the Microsoft Data Access SDK on the <u>Microsoft Web site</u>.

All versions of the Microsoft Foundation Classes database classes are ODBC version 2.*x* compliant. 2.*x* versions of the ODBC header files Sql.h, Sqlext.h, and Sqltypes.h ship in the Include directory of Microsoft Visual C++. You must ensure that 3.*x* versions of the header files are included in the build process and that 3.*x* versions of the libraries are used for linking.

Sample Data

Some of the samples rely on sample data provided in the C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\Odbc\Data directory.

Three directories contain data for the samples. The BCP and LoadData samples each have a copy of the Microsoft Access-developed **Northwind** example data.

Directory	Description
\Data\Northbcp	Contains a copy of the Northwind data
\Data\North	Contains a copy of the Northwind data
\Data\Trans	Contains data used by the transaction processing
	and concurrency control sample

The sample data can be installed into any database. A Transact-SQL script, Createtb.sql, is installed into each sample data directory. The script creates the sample tables and stored procedures.

The scripts drop the tables they create so that they can be run multiple times as an example. Running the scripts in a database that contains data other than the supplied sample data can cause unintended results.

The sample data in \Data\Trans can be installed in any database. The sample data

in the other two directories can be installed in any database except the Northwind sample database installed with SQL Server.

To load the sample data

- First make the ODBC samples and copy these files to a directory in your computer's path: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Sa C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Sa
- 2. Open Control Panel/ODBC and define an ODBC data source with the database you want to hold the sample data as the default database.
- 3. Open a command prompt window: cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools

When loaddata.exe starts, connect to the data source defined in step 2. Open the command file lddist.cmd using the CommandFile window, and then click **GO**. Click **Close** when the commands complete.

In the command prompt window:
 cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools

When loaddata.exe starts, connect to the data source defined in step 2. Open the command file ldnorthw.cmd using the CommandFile window, and then click **GO**. Click **Close** when the commands complete.

5. As an alternative to step 4 you can: cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools

When BCPSamp.exe starts, connect to the data source defined in step 2. Open the command file bcpnorth.cmd using the CommandFile window, and then click **GO**. Click **Close** when the commands complete.

See Also

Samples

Cursors and Transactions, Data Entry and Concurrency

The cursor sample illustrates using Microsoft® SQL Server[™] cursors to allow concurrent access to data on a server running an instance of SQL Server. The sample shows three methods of concurrent data access:

- Optimistic, using SQL Server **timestamp** data types.
- Pessimistic, using extremely isolated transactions and row-level locking.
- Pessimistic, using the Microsoft Distributed Transaction Coordinator (MS DTC).

A dialog box, available when no document windows are open, allows the user to select the method of concurrency control to be used by the application.

The sample relies on data shipped with the ODBC 3.*x* samples. The data can be installed using the LoadData sample and the Lddist.cmd file. For more information about Lddist.cmd, see <u>ODBC Samples</u>.

The sample is a Microsoft Foundation Class MDI application. Each MDI window contains a property sheet of customer data from a fictitious bank. The window is implemented on a separate connection to the database so that a single user can experiment with various concurrency handling methods from a single instance of the application.

Customers at the bank can have one or two accounts and you can either deposit or withdraw funds from a customer's checking or savings account. Each transaction inserts a row into an activity file and then attempts a positioned update of a balance item for the account. If the balancing transaction fails, the activity file insert is rolled back.

To open a connection, select **File****New** from the menu.

SQLAllocHandle	SQLDriverConnect	SQLGetDiagRec
SQLBindParameter	SQLEndTran	SQLNumResultCols
SQLCloseCursor	SQLExecDirect	SQLSetConnectAttr
SQLColAttribute	SQLFetch	SQLSetEnvAttr
SQLDataSources	SQLFreeHandle	SQLSetPos
SQLDescribeCol	SQLGetCursorName	SQLSetStmtAttr
SQLDisconnect	SQLGetDiagField	

LoadData

The LoadData sample illustrates using **SQLPrepare** and **SQLExecute** to insert large amounts of data into Microsoft® SQL Server[™] tables.

LoadData is a general-purpose utility for loading data not bound by native data formats or character restrictions onto a server running an instance of SQL Server.

The sample illustrates:

- Using arrays of parameters for rapid execution of RPC batches.
- Using manual-commit mode to break batches into units of work.
- Using data-at-execution parameters to insert values into SQL Server **text** and **image** columns.

To build the application, you must ensure that 3.*x* versions of the ODBC header files and libraries are used, and that the SQL Server 2000 version of Odbcss.h is used.

The sample is a Microsoft Foundation Class dialog application. The application allows you to connect to a defined ODBC SQL Server data source and requires that you enter the name of a command file to process.

The command files of the LoadData sample application allow the user to tailor command processing by using the application. The application recognizes two commands:

- ScriptRun, which processes Transact-SQL statements.
- LoadData, which executes a prepared INSERT statement with parameters to copy data to a server running SQL Server.

The application treats strings enclosed in brackets ([]) as progress text and displays them in its progress pane as the application processes a command file.

Sample data, containing a command file, script file, and ANSI text data files are

included.

Command File Syntax

[text] ScriptRun "file_name" LoadData "database..table", "file_name"

Arguments

[text]

Is progress text. Text between the enclosing brackets is displayed in the dialog box within its progress group.

ScriptRun "file_name"

Attempts to open and read the text file indicated in the *file_name* parameter. The text file must contain ODBC or Transact-SQL. The application processes multiple lines of text as a single batch, using **SQLExecDirect** to execute the SQL batch when the string "go" is located on a single line of the file.

LoadData "database..table", "file_name"

Copies data from the client file, specified in the *file_name* parameter, to the SQL Server table specified in the *database..table* parameter.

To run the loaddata sample

- Build a .cmd file containing ScriptRun and LoadData commands. For an example, see the file Ldnorthw.cmd in this directory: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Sa
- Run the sample by changing to the directory with the command file and specifying the samples name on the command prompt: cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools Loaddata

When Loaddata.exe starts, connect to an ODBC data source, specify the location of the command file in the CommandFile window, and then click **GO**.

SQLAllocHandle	SQLDriverConnect	SQLParamData
SQLBindParameter	SQLExecDirect	SQLPrepare
SQLCloseCursor	SQLExecute	SQLPutData
SQLColAttribute	SQLFreeHandle	SQLSetConnectAttr
SQLDataSources	SQLGetDiagField	SQLSetEnvAttr
SQLDescribeCol	SQLGetDiagRec	SQLSetStmtAttr
SQLDisconnect	SQLNumResultCols	

Performance

The MFCPerf sample illustrates two things: tuning the MFC ODBC database classes for performance, and capturing and interpreting Microsoft® SQL Server[™] ODBC driver performance data.

The sample uses advanced features of the MFC ODBC database classes and must be built with MFC version 4.2 or later. Although MFC is ODBC 2.x compliant, the MFCPerf sample uses features of the SQL Server ODBC driver available only in SQL Server 2000. The application must be built with ODBC 3.x header files and libraries.

The sample is an MFC MDI application that uses the **Northwind** sample database.

The application includes three documents. Two of the documents share a single view that displays customer order history. These documents are selected using the **File\New Fast** or **File\New Slow** menu items. One of the documents is identified as "slow." It uses MFC filter strings to execute SELECT statements, retrieving data from the **Northwind Orders** and **OrderDetails** tables. The second document, identified as "fast," uses parameterized execution of a SQL Server stored procedure to accomplish the same task.

The **PerfTest** menu, available when no document windows are open, automates creation of one of each document and steps through 20 client records. The process captures the SQL Server performance statistics for each document and displays them. The "fast" document shows a dramatic decrease in server roundtrips and a corresponding drop in data moved across the network.

SQLAllocHandle	SQLSetConnectAttr	CRecordSet::Requery
SQLDataSources	CDatabase::Open	CRecordSet::MoveNext
SQLGetConnectAttr	CRecordSet::Open	CRecordSet::IsEOF

ODBC Bulk Copy Sample

The ODBC bulk copy sample illustrates using Microsoft® SQL Server[™] bulk copy functions with the SQL Server ODBC driver.

To build the application, you must ensure that 3.*x* versions of the ODBC header files and libraries are used, that the SQL Server 2000 version of Odbcss.h is used, and that the linker can find Odbcbcp.lib.

The sample is a Microsoft Foundation Class dialog application. The application allows you to connect to a defined ODBC SQL Server datasource and requires that you enter the name of a command file to process.

Command files allow you to tailor command processing by the application. The application recognizes two commands: ScriptRun, which processes Transact-SQL statements; and BCPData, which performs a bulk copy operation. The application treats strings enclosed in brackets ([]) as progress text and displays them in its progress panel area as the application processes a command file.

Sample data, containing a command file, script file, and character format **bcp** data files for the **Northwind** sample database are included.

Command File Syntax

[text]
ScriptRun "file_name"
BCPData "database..table", "file_name", "errorfile", direction

Arguments

[text]

Is progress text. Text between the enclosing brackets is displayed in the dialog box within its progress group.

ScriptRun "file_name"

Attempts to open and read the text file indicated in the *file_name* parameter. The text file must contain ODBC or Transact-SQL. The application processes multiple lines of text as a single batch, using **SQLExecDirect** to

execute the SQL batch when the string "go" is located on a single line of the file.

BCPData "database..table", "file_name", "errorfile", direction

Performs a **bcp** command to copy data to or from a SQL Server table. The *database..table* and *file_name* parameters are required and specify the server table and the local file respectively. The *errorfile* parameter can be an empty string. No error logging is performed if it is. The *direction* parameter must be either IN or OUT.

To run the ODBC bulk copy sample

- Build a .cmd file containing ScriptRun and DBCData commands. For an example, see the file Bcpnorth.cmd in this directory: C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Se
- Run the sample by changing to the directory with the command file and specifying the samples name on the command prompt: cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools Bcpsamp
- 3. When Bcpsamp.exe starts, connect to an ODBC data source, specify the location of the command file in the CommandFile window, and then click **GO**.

Bcp_control	SQLDisconnect	SQLNumResultCols
Bcp_exec	SQLDriverConnect	SQLPrepare
Bcp_init	SQLExecDirect	SQLSetConnectAttr
SQLAllocHandle	SQLFreeHandle	SQLSetEnvAttr
SQLColAttribute	SQLGetDiagField	
SQLDataSources	SQLGetDiagRec	

COMPUTE Clause and Multiple Result Sets

The compute sample illustrates handling the multiple result sets that occur when an application executes a Transact-SQL SELECT statement containing a COMPUTE clause.

The sample executes a statement, and then uses **SQLMoreResults** and **SQLColAttribute** to determine the shape of each result set generated.

The sample shows how to mix array and single-row binding for rapid and spaceefficient handling of the output.

Before compiling the compute sample, open the project and locate these lines in compute.cpp:

PTSTR	<pre>szDataSource = _T("MyDatasource");</pre>
PTSTR	szUID = _T("MyUID");
PTSTR	szPWD = _T("MyPwd");

Replace the strings:

- MyDatasource with the name of an ODBC datasource that has the Northwind sample database as its default database.
- MyUID with a valid login ID.
- MyPwd with the password for the login specified for szUID.

After compiling and linking the sample, run it by specifying its name at the command prompt:

cd C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Sample: Compute

SQLAllocHandle SQLDisconnect SQLNumResultCols	SQLAllocHandle
---	----------------

SQLBindCol	SQLExecDirect	SQLSetEnvAttr
SQLColAttribute	SQLFetch	SQLSetStmtAttr
SQLConnect	SQLFreeHandle	
SQLDescribeCol	SQLGetDiagRec	